Oracle® Application Server Containers for J2EE

Support for JavaServer Pages Developer's Guide

10*g* Release 2 (10.1.2)

Part No. B14014-01

November 2004

ORACLE®

Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide, 10*g* Release 2 (10.1.2)

Part No. B14014-01

Primary Author: Dan Hynes, Brian Wright

Contributing Author: Michael Freedman

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

Contents

Send Us Your Comments

Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide, 10*g* Release 2 (10.1.2)

Part No. B14014-01

Oracle welcomes your comments and suggestions on the quality and usefulness of this publication. 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 about this manual?

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

- Electronic mail: appserverdocs_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 electronic mail address (optional).

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 an industry consortium led by Sun Microsystems. It summarizes standard features but focuses primarily on Oracle implementation details and value-added features. An overview of standard JSP technology is followed by discussion of the OC4J implementation, JSP configuration, basic programming considerations, JSP strategies and tips, translation and deployment, JSP tag libraries, and globalization support.

JavaServer Pages technology is a component of the standard Java 2 Enterprise Edition (J2EE). The J2EE component of the Oracle Application Server is known as the Oracle Application Server Containers for J2EE (OC4J).

The OC4J JSP container in Oracle Application Server 10*g* Release 2 (10.1.2) is a complete implementation of the Sun Microsystems *JavaServer Pages Specification, Version 1.2*.

This preface contains the following sections:

- Intended Audience
- Documentation Accessibility
- Structure
- Related Documents
- Conventions

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 Oracle Database)

While some information about standard JSP 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 features, consult the Sun Microsystems *JavaServer Pages Specification* 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.

For documentation of tag libraries and utilities that are provided with the OC4J product, please refer to the *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference*.

For a quick primer about getting started with JSP pages in OC4J, see the *Oracle Application Server 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 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 does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Structure

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 XML support, primarily added in the JSP 1.2 specification. JSP XML syntax and the JSP XML view are described.

Chapter 6, "Additional Programming 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 OC4J JSP translator and Oracle `ojspc` pretranslation 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, "Third Party Licenses"

This appendix includes the Third Party License for third party products included with Oracle Application Server and discussed in this document.

Related Documents

For more information, see these Oracle resources available from the Oracle Java Platform Group:

- *Oracle Application Server 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.

- *Oracle Application Server Containers for J2EE Stand Alone User's Guide*

  This version of the user's guide is specifically for the standalone version of OC4J, and is available when you download the standalone version from OTN. OC4J standalone is used in development environments, but not typically in production environments.

- *Oracle Application Server Containers for J2EE Servlet Developer's Guide*

This book provides information for servlet developers regarding use of servlets and the servlet container in OC4J, including basic servlet development, use of JDBC and EJBs, building and deploying applications, and servlet and Web site configuration. Consideration is given to both OC4J in a standalone environment for development and OC4J in Oracle Application Server for production.

- *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference*

  This book provides conceptual information and detailed syntax and usage information for tag libraries, JavaBeans, and other Java utilities provided with OC4J. There is also a summary of tag libraries from other Oracle product groups.

- *Oracle Application Server Containers for J2EE Services Guide*

  This book provides information about standards-based Java services supplied with OC4J, such as JTA, JNDI, JMS, JAAS, and the Oracle Application Server Java Object Cache.

- *Oracle Application Server Containers for J2EE Security Guide*

  This document (not to be confused with the *Oracle Application Server 10g Security Guide*), describes security features and implementations particular to OC4J. This includes information about using JAAS, the Java Authentication and Authorization Service, as well as other Java security technologies.

Also available from the Oracle Java Platform group:

- *Oracle Database Java Developer's Guide*
- *Oracle Database JDBC Developer's Guide and Reference*
- *Oracle Database JPublisher User's Guide*

Available from the Oracle Application Server group:

- *Oracle Application Server Administrator's Guide*
- *Oracle Application Server Security Guide*
- *Oracle Application Server Performance Guide*
- *Oracle Enterprise Manager Concepts*
- *Oracle HTTP Server Administrator's Guide*
- *Oracle Application Server Globalization Guide*
- *Oracle Application Server Web Cache Administrator's Guide*
- *Oracle Application Server Web Services Developer's Guide*
- *Oracle Application Server Upgrading to 10g Release 2 (10.1.2)*

Available from the Oracle JDeveloper group:

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

  `http://www.oracle.com/technology/products/jdev/content.html`

Available from the Oracle Server Technologies group:

- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle XML API Reference*
- *Oracle Database Application Developer's Guide - Fundamentals*

- *PL/SQL Packages and Types Reference*
- *PL/SQL User's Guide and Reference*
- *Oracle Database SQL Reference*
- *Oracle Database Net Services Administrator's Guide*
- *Oracle Advanced Security Administrator's Guide*
- *Oracle Database Reference*

Printed documentation is available for sale in the Oracle Store at

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

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://www.oracle.com/technology/membership/`

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://www.oracle.com/technology/documentation`

The following OTN Web site for Java servlets and JavaServer Pages is also available:

`http://www.oracle.com/technology/tech/java/servlets/`

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

The following conventions are also used in this manual:

Convention	Meaning
. . .	Horizontal ellipsis points in statements or commands mean that parts of the statement or command not directly related to the example have been omitted
boldface text	Boldface type in text indicates a term defined in the text, the glossary, or in both locations.

Convention	Meaning
Italics	Italic typeface indicates book titles or emphasis, or terms that are defined in the text.
`Monospace (fixed-width) font`	Monospace typeface within text indicates items such as executables, file names, directory names, Java class names, Java method names, variable names, other programmatic elements (such as JSP tags or attributes, or XML elements or attributes), or database SQL commands or elements (such as schema names, table names, or column names).
`Italic monospace (fixed-width) font`	Italic monospace font represents placeholders or variables.
< >	Angle brackets enclose user-supplied names.
[ ]	Brackets enclose optional clauses from which you can choose one or none.
|	A vertical bar represents a choice of two or more options. Enter one of the options. Do not enter the vertical bar.

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*.

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

For an overview of the JSP implementation in Oracle Application Server Containers for J2EE (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 chapter contains the following sections:

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

Note: The Sample Applications chapter available in previous releases has been removed. Applications that were listed there are available in the OC4J demos, available from the following location on the Oracle Technology Network (requiring an OTN membership, which is free of charge):

`http://www.oracle.com/technology/tech/java/oc4j/demos/`

Introduction to JavaServer Pages

JavaServer Pages 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.

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.

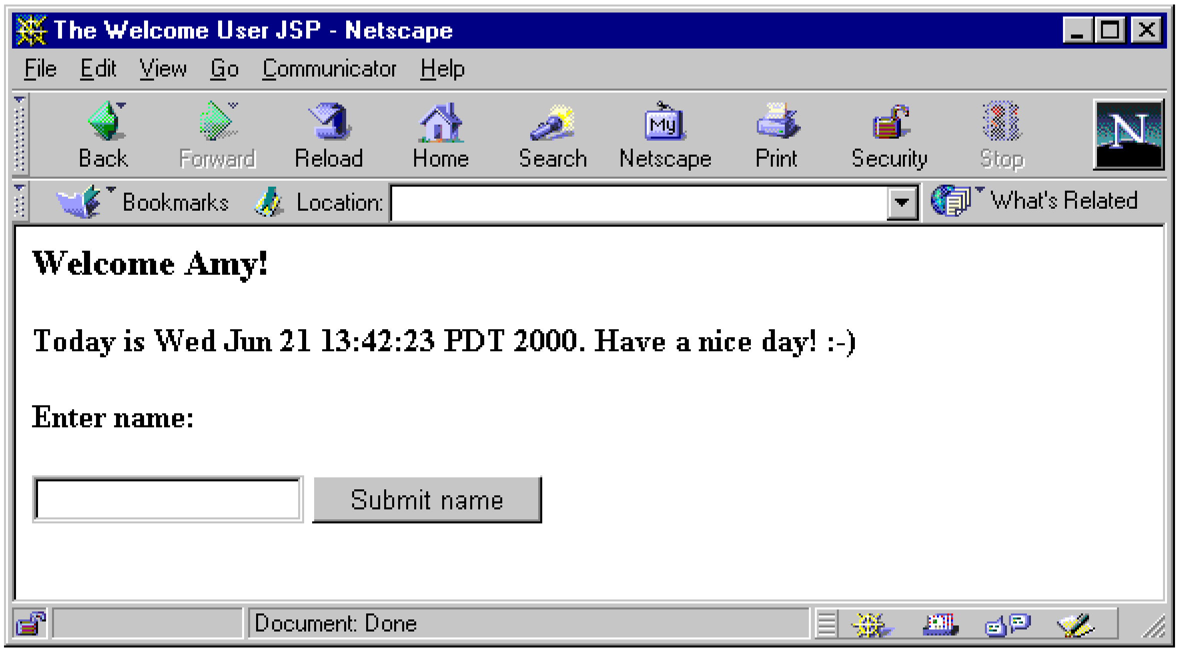
What a JSP Page Looks Like

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 1-5.

```
<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 5-3.) In this example, Java snippets get the user name from an HTTP request object, print the user name, and get the current date.

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


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

Welcome Amy!

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

Enter name:

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-2 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 3-7 for more information. Therefore, beginning with Oracle9*i*AS Release 2 (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-8) 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 might 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 1-12.

JSP Pages and Alternative Markup Languages

JavaServer Pages technology is typically used for dynamic HTML output, but the JSP specification 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 database 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 *Oracle Application Server 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*.

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 specification supports the following directives:

- `page`
- `include`
- `taglib`

page directive

Use this directive to specify any of a number of page-dependent attributes, such as scripting language, content type, character encoding, 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" %>
```

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

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

This example unbuffers 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.
- 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 9-1.

include directive

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 1-21 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 6-1.
- The `include` directive can be used only between files in the same servlet context (application).
- See "JSP File Naming Conventions" on page 3-7 for information about naming conventions for included files.

taglib directive

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 1-18, 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
- Expressions
- Scriptlets
- Comments

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 6-7 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 normal Java syntax.

Example 1:

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

Three one-line JSP scriptlets are intermixed with two lines of HTML 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 6-7.

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.

These topics are discussed in the following sections:

- Explicit Objects
- Implicit Objects
- Using an Implicit Object
- Object Scopes

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 1-12, 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 1-11.

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

http://java.sun.com/products/servlet/2.3/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` interface.

- `application`

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

  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, which 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 4-14.

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 1-9:

```
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 can 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>
```

Alternatively, if 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 immediately below:

- `jsp:usebean`
- `jsp:setProperty`
- `jsp:getProperty`
- `jsp:param`

- `jsp:include`
- `jsp:forward`
- `jsp:plugin`

jsp:useBean tag

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 1-18. Scopes are discussed in "JSP Objects and Scopes" on page 1-9.

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.

The following 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="reqobj" type="mypkg.MyIntfc" scope="request" />
```

This next example uses a page-scope instance `pageobj` of class `PageBean`, first creating it if necessary:

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

The following example creates an instance of class `SessionBean` and assigns the instance to the variable `sessobj` of type `MyIntfc`:

```
<jsp:useBean id="sessobj" class="mybeans.SessionBean"
```

```
              type="mypkg.MyIntfc scope="session" />
```

jsp:setProperty tag

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 1-17.

Important: Note the following for `property="*"`:

- To specify that iteration should continue if an error is encountered, set the `setproperty_onerr_continue` configuration parameter to `true`. This parameter is described under "JSP Configuration Parameters" on page 3-11.
- The JSP 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, 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 tag

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 tag

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 tag

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 specification and the OC4J JSP container support either a "`true`" or "`false`" setting, with "`false`" being the default. (The JSP 1.1 specification supported 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 6-1.
- The `jsp:include` tag can be used only between pages in the same servlet context (application).

jsp:forward tag

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 and not output 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 tag

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 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 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, covered in the following sections:

- 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
Character or char	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
Integer or int	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*.

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 1-6.

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

- 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 in the standard `javax.servlet.jsp.tagext` package. Which interface to implement depends on whether the tag has a body and whether the tag handler requires access to the body content.

- 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* 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*.

For complete information about the tag libraries provided with OC4J, see the *Oracle Application Server 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-2.

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 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 3-10.

Pretranslation Model

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

- It can save time for the users when they first request a JSP page, because translation at execution time is not necessary.
- It is 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 Pretranslation" on page 7-26 and "Deployment of Binary Files Only" on page 7-28.

Oracle supplies the `ojspc` command-line utility for pretranslating 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 Pretranslation Utility" on page 7-8.

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-8). 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 might 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 actually 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 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` context path 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 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: General servlet and JSP invocation are discussed in the *Oracle Application Server 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 1-12 discusses the `jsp:include` and `jsp:forward` statements.)

2 Overview of the Oracle JSP Implementation

The JSP container provided with Oracle Application Server Containers for J2EE (OC4J) in the Oracle Application Server is a complete implementation of the JSP 1.2 specification. This functionality depends upon servlet 2.3 functionality, and the OC4J servlet container is a complete implementation of the servlet 2.3 specification.

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

The following sections are included:

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

Overview of the Oracle Application Server and JSP Support

The following sections provide a brief overview of the Oracle Application Server, its J2EE environment, its JSP implementation, and its Web server:

- Overview of the Oracle Application Server
- Oracle HTTP Server and mod_oc4j
- Overview of OC4J
- Overview of the JSP Implementation in OC4J

Note: Users of earlier Oracle Application Server releases can refer to *Oracle Application Server Upgrading to 10g Release 2 (10.1.2)* for information about issues in migrating to the current release.

Overview of the Oracle Application Server

Oracle 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 services, and management and security. For performance, scalability, and dependability, there are also clustering and load-balancing features.

To deliver this range of content and services, the Oracle Application Server incorporates many components, including the Oracle HTTP Server, Oracle Application Server Web Cache, Oracle Application Server Web Services, Oracle Application Server Portal, Oracle Application Server Wireless, and business logic runtime environments that support Enterprise JavaBeans, stored procedures, and Oracle Application Development Framework (Oracle ADF) Business Components.

For its Java environment, Oracle Application Server provides the Oracle Application Server Containers for J2EE (OC4J), a J2EE 1.3-compliant set of containers and services. This includes the JSP container described in this manual, a servlet container, and an EJB container.

For administration, you can fully manage and configure Oracle Application Server and OC4J using the HTML-based Oracle Enterprise Manager 10*g*. This includes full support for managing clustering, configuration, and deployment.

Oracle HTTP Server and mod_oc4j

Oracle HTTP Server, powered by the Apache Web server, is included with Oracle 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.

Oracle Application Server 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.

Note: Oracle HTTP Server is not relevant for an OC4J standalone environment (typically used only during development). See "OC4J Standalone" on page 2-5.

Overview of OC4J

The following sections provide an overview of features of OC4J, the J2EE component of the Oracle Application Server:

- OC4J General Features

- OC4J Services
- OC4J Containers
- OC4J Standalone

OC4J General Features

OC4J is a high-performance, J2EE-compliant set of containers and services providing a scalable and reliable server infrastructure. In Oracle Application Server 10*g* Release 2 (10.1.2), OC4J complies with the J2EE 1.3 specification.

For developer convenience, OC4J has been integrated with Oracle JDeveloper and other development tools, and can run in a standalone mode separate from Oracle Application Server during the development process.

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

For security, OC4J supports Secure Socket Layer (SSL) and HTTPS functionality.

Note: Each OC4J instance runs in a single Java virtual machine.

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.
- JDBC data sources: This is the standard mechanism for connecting to a database.

See the *Oracle Application Server Containers for J2EE Services Guide* for information.

OC4J Containers

The OC4J 10.1.2 implementation 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 2-5.

- A servlet container complying with the servlet 2.3 specification (with key features described immediately below)
- An EJB container complying with the EJB 2.0 specification (with key features described below)

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 the following key features:

- SSL and HTTPS: In Oracle Application Server, OC4J supports SSL (Secure Socket Layer) communication between Oracle HTTP Server and OC4J, using secure AJP. In addition, OC4J standalone supports SSL communication directly between a client and OC4J, using HTTPS.
- Integration with SSO and Oracle Internet Directory: This is through the Oracle JAAS implementation.
- Stateful failover and cluster deployment: For a distributable application, session state is replicated to alternate OC4J servers so that state is not lost in the event of failover.
- 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.

See the *Oracle Application Server Containers for J2EE Servlet Developer's Guide* for information.

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

See the *Oracle Application Server Containers for J2EE Enterprise JavaBeans Developer's Guide* for information.

OC4J Standalone

In a production environment, it is typical to use OC4J inside a complete Oracle Application Server environment, including the Oracle HTTP Server (as described in "Oracle HTTP Server and mod_oc4j" on page 2-2), OracleAS 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 at the following location:

`http://www.oracle.com/technology/tech/java/oc4j`

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

> **Note:** To use OC4J standalone, you must have a supported version of the Sun Microsystems JDK installed. A JDK is not provided with the OC4J standalone product.

Overview of the JSP Implementation in OC4J

The JSP container in Oracle Application Server 10*g* Release 2 (10.1.2) is compliant with the JSP 1.2 specification.

In general, a JSP 1.2 environment requires a servlet 2.3 environment, such as the OC4J servlet container. The OC4J JSP implementation, however, also supports running on 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.

These features are discussed in the following sections:

- 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. 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 3-10.

For non-OC4J environments, use the old front-end servlet, `oracle.jsp.JspServlet`.

OC4J JSP Features for JSP 1.2

In the OC4J 10.1.2 implementation, the OC4J JSP container is fully compliant with the JSP 1.2 specification. Most functionality introduced in this specification is in the area of custom tag libraries.

- Tag library features:
  - There is a 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 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 8-3 for a more detailed summary.

- XML features:
  - The OC4J JSP container previously supported a standard XML-alternative syntax, but this is replaced with newer technology according to the current JSP 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:

  The OC4J JSP implementation 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 9-1.

Configurable JSP Extensions in OC4J

In addition to JSP 1.2 compliance, the OC4J JSP container in Oracle Application Server 10*g* Release 2 (10.1.2) includes the following notable features.

Also see "Overview of Oracle Value-Added Features" on page 2-9.

The following have been supported since 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

  The JSP 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 Parameter Descriptions" on page 3-13, see the description of the `forgive_dup_dir_attr` parameter.

- Separate mode switches for XML validation of `web.xml` file and TLD files

  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 Parameter Descriptions" on page 3-13, see the descriptions of the `xml_validate` and `no_tld_xml_validate` parameters.

- Mode flag for extra imports

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

  In "JSP Configuration Parameter Descriptions" on page 3-13, see the description of the `extra_imports` parameter.

- "Well-known" location for sharing tag libraries

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

  In "JSP Configuration Parameter Descriptions" on page 3-13, see the description of the `well_known_taglib_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. In "JSP-Related OC4J Configuration Parameter Descriptions" on page 3-20, see the description of the `jsp-timeout` parameter.

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

- Mode switch for automatic page retranslation and reloading

  You have a choice of: 1) running JSP pages without any automatic reloading or retranslation of JSP pages; 2) automatically reloading any page implementation classes (but not JavaBeans or other dependency classes); or 3) automatically retranslating any JSP pages that have changed.

In "JSP Configuration Parameter Descriptions" on page 3-13, see the description of the `main_mode` parameter.

- Tag handler instance pooling

  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 8-30.

- Output mode for null output

  You can print an empty string instead of the default "null" string for null output from a JSP page.

  In "JSP-Related OC4J Configuration Parameter Descriptions" on page 3-20, see the description of the `jsp-print-null` parameter.

- Single-threaded-model JSP instance pooling

  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. (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 servlet context and HTTP session objects.

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

Oracle JDeveloper JSP Support

Visual Java programming tools now typically support JSP coding. In particular, Oracle 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 Oracle ADF Business Components, offered with JDeveloper

See "Application Deployment with Oracle JDeveloper" on page 7-25 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://www.oracle.com/technology/products/jdev/content.html`

(You will need an Oracle Technology Network membership, which is free of charge.) For an overview of JSP tag libraries provided with JDeveloper, see the *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Note: Other key IDE vendors have built plug-in modules that allow seamless integration with OC4J. This provides developers with the capability to build, deploy, and debug J2EE applications running on OC4J directly from within the IDE. You can refer to the following Web site for more information:

`http://www.oracle.com/technology/products/ias/9ias_partners.html`

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 summaries and overviews in 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 *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Summary of Tag Libraries and Utilities Provided with OC4J

This section provides a brief summary of extended OC4J JSP features that are implemented through standards-compliant custom tag libraries, custom JavaBeans, and other classes. These features are documented in the *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference*:

- Extended types implemented as JavaBeans that can have a specified scope
- `JspScopeListener` for event-handling
- Integration with XML and XSL
- Data-access tag library (sometimes referred to as "SQL tags") and JavaBeans
- The JSP Markup Language (JML) custom tag library, which reduces the level of Java proficiency required for JSP development

- Web services tag library
- Tag libraries and JavaBeans for uploading files, downloading files, and sending e-mail from within an application
- EJB tag library
- Additional utility tags, such as for displaying dates and currency amounts appropriately for a specified locale

Note: See "Overview of Tags and API for Caching Support" on page 2-11 for an overview of additional tag libraries for caching.

Overview of Oracle-Specific Features

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

- *Global includes*, a mechanism to automatically statically include a file or files in multiple pages
- Dynamic Monitoring Service (DMS) support for performance measurements
- Enhanced application framework and globalization support for servlet 2.0 environments

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 7-6.

Support for Dynamic Monitoring Service

DMS adds performance-monitoring features to a number of Oracle Application Server 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). Times are in milliseconds.

- 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 `application.xml` and `default-web-site.xml` configuration file. Use the Enterprise Manager to access DMS, display DMS information, and, if appropriate, alter DMS configuration.

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

Enhanced Servlet 2.0 Support

OC4J supports special features for a servlet 2.0 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
- Extended globalization support for servlet 2.0 environments

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

  The OracleAS Web Cache provides an ESI engine.

- A tag library 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 Oracle Application Server Java Object Cache as its default repository.

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

Support for the JavaServer Pages Standard Tag Library

With Oracle Application Server 10*g* Release 2 (10.1.2), 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 access to 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 organized into four JSTL sublibraries according to these functional areas.

For a more complete summary of JSTL support, you can refer to the *Oracle Application Server 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 data-access (SQL) tag libraries provided with OC4J pre-date JSTL and have areas of duplicate functionality. Going forward, for standards compliance, it is generally advisable to use JSTL instead of the custom libraries. 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 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 sections are included in this chapter:

- Some Initial Considerations
- Key Support Files Provided with OC4J
- JSP Configuration in OC4J
- Key OC4J Configuration Files
- JSP Configuration in Oracle Enterprise Manager 10g

Note: JSP pages will run with any standard browser supporting HTTP 1.0 or higher. The JDK or other Java environment in the user's Web browser is irrelevant, because all the Java code in a JSP page is executed in the Web server.

Some Initial Considerations

The following sections discuss some considerations you should be aware of before you begin coding or using JSP pages:

- Application Root Functionality
- Classpath Functionality
- Runtime Retranslation or Reloading
- JSP Compilation Considerations
- JSP Security Considerations
- JSP Performance Considerations
- Default Package Imports
- JSP File Naming Conventions

- JDK 1.4 Considerations: Cannot Invoke Classes Not in Packages

Application Root Functionality

The servlet specification (since servlet 2.2) provides 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 standard servlet 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 document 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 *Oracle Application Server 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 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.

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 (`.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 and depends on the value of the `jsp-cache-directory` configuration parameter. See "JSP Translator Output File Locations" on page 7-5 for information.

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

Runtime Retranslation or Reloading

During runtime, any retranslation of JSP pages or reloading of JSP page implementation classes is controlled by the JSP `main_mode` configuration parameter. Possible settings are `recompile` (default) to retranslate JSP pages that have changed, `reload` to reload classes that were generated by the JSP container and have changed (such as page implementation classes), or `justrun` to run without any timestamp-checking, for optimal performance in production environments. See "JSP Configuration Parameter Descriptions" on page 3-13 for additional information.

Notes:

- This discussion is not relevant for pretranslation scenarios.
- The OC4J JSP container does not have its own classloader.
- 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.

For information about classloading behavior at the servlet layer, see the *Oracle Application Server Containers for J2EE Servlet Developer's Guide*.

JSP Compilation Considerations

Java compilation can be either *in-process*, running in the same process as OC4J, or *out-of-process*, running in a separate process.

By default, OC4J as a whole uses out-of-process compilation, and the compiler is invoked as a separate executable. The default compiler executable is `javac` from the Sun Microsystems JDK; however, you can configure OC4J to use a different compiler. In OC4J standalone, you can accomplish this by adding a `<java-compiler>` element, with desired settings, to the OC4J `server.xml` file. In an Oracle Application Server environment, use Oracle Enterprise Manager 10*g* to change this configuration.

For improved JSP performance, however, the OC4J JSP container uses in-process compilation by default, assuming that the `tools.jar` file of the Sun Microsystems JDK is installed and in the classpath. With in-process compilation, the compiler class is invoked directly. You can use a `<library>` element in the `server.xml` file to ensure that `tools.jar` is in the classpath.

There are also two related JSP configuration parameters: `use_old_compiler` and `javaccmd`:

- You can set `use_old_compiler` to `false` to force the JSP container to use the same compiler as the rest of OC4J—out-of-process compilation with `javac` by default, or compilation according to a `<java-compiler>` element in `server.xml`. The `use_old_compiler` flag is set to `true` by default if `tools.jar` is in the classpath, resulting in in-process compilation unless `javaccmd` is set.
- If you want to use an out-of-process compiler, but not the compiler that the rest of OC4J uses, then set `use_old_compiler` to `true` and use the `javaccmd` parameter to specify the desired compiler. (The `javaccmd` parameter is ignored if `use_old_compiler` is set to `false`.)

Notes:

- If `tools.jar` is not in the classpath, then `use_old_compiler` is forced to a `false` setting.
- The `use_old_compiler` and `javaccmd` parameters are further discussed under "JSP Configuration Parameter Descriptions" on page 3-13.
- See the *Oracle Application Server Containers for J2EE User's Guide* for information about elements of the `server.xml` file.

JSP Security Considerations

With respect to application security, be aware that you should 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 Parameter Descriptions" on page 3-13.

JSP Performance Considerations

The following sections summarize JSP, OC4J, and Oracle Application Server features for performance optimization and monitoring:

- Programmatic Considerations for Optimization
- Configuration Optimizations
- The ojspc Utility for Pretranslation
- Additional OC4J and Oracle Application Server Performance Features

Programmatic Considerations for Optimization

You might consider the following when creating your JSP pages:

- Unbuffer JSP pages. 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.

- Avoid using HTTP session objects if they are not required. If a JSP page does not require an HTTP session (essentially, does not require storage or retrieval of session attributes), then you can specify that no session is to 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-3.

- In addition to general Oracle Application Server caching features such as OracleAS Web Cache and Java Object Cache, there are caching features that are specific to JSP pages and are available through custom tag libraries provided with OC4J. For a brief overview of these features—the JESI tag library and the Web Object Cache—see "Overview of Tags and API for Caching Support" on page 2-11. For additional information, refer to the *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference.*

Configuration Optimizations

There are a number of JSP and OC4J configuration parameters that affect performance:

- In a production environment, where JSP pages do not change, you should configure the JSP container to *not* check timestamps (which it would otherwise do to see if any pages require retranslation). You can specify this by setting the Oracle JSP configuration parameter `main_mode` to the value `justrun`. See "JSP Configuration Parameter Descriptions" on page 3-13 for information about this parameter.
- You can also improve performance with tag libraries by specifying that tag handler instances be reused within each JSP page. For optimal results, especially for JSP pages with very large numbers of custom tags, specify that the logic and patterns of tag handler reuse be determined at translation time instead of runtime. You can specify this through the Oracle JSP configuration parameter `tags_reuse_default`. See "Disabling or Enabling Runtime or Compile-Time Tag Handler Reuse" on page 8-30.
- There are a number of additional Oracle JSP configuration parameters that can affect performance, either favorably or unfavorably. See "JSP Configuration Parameter Descriptions" on page 3-13 for information about `check_page_scope`, `precompile_check`, `reduce_tag_code`, and `static_text_in_chars`.
- Features have been added to make tag library usage more efficient. The key feature is persistent caching for tag library descriptor (TLD) files, which you can enable through the OC4J configuration parameter `jsp-cache-tlds`. See "Oracle Extensions for Tag Library Sharing and Persistent TLD Caching" on page 8-15.
- The OC4J configuration parameters `simple-jsp-mapping` and `enable-jsp-dispatcher-shortcut` can significantly affect performance. See "JSP-Related OC4J Configuration Parameter Descriptions" on page 3-20 for information about these parameters.

The ojspc Utility for Pretranslation

You might consider using the `ojspc` utility to pretranslate JSP pages. This avoids the performance cost of translating pages as they are first accessed by users. See "JSP Pretranslation" on page 7-26 for additional discussion of the advantages of pretranslation. See "The ojspc Pretranslation Utility" on page 7-8 for details about the utility itself.

Additional OC4J and Oracle Application Server Performance Features

Note the following OC4J and Oracle Application Server features for performance optimization and monitoring:

- OC4J JSP code for output `JspWriter` objects using nondefault character sets has been optimized.
- Tag libraries provided with OC4J, such as the data-access (SQL) tag library, are optimized to take advantage of additional Oracle resource pooling and resource cleanup features, such as for database connections.

- You can use the Oracle Application Server Dynamic Monitoring Service to track performance. See "Support for Dynamic Monitoring Service" on page 2-10.
- For information about general OC4J or Oracle Application Server features for performance and robustness, consult the *Oracle Application Server Containers for J2EE User's Guide* and the *Oracle Application Server Performance Guide*.

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 earlier 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 1-6, and see "Page Directive import Settings Are Cumulative" on page 6-10.

  For multiple pages, you can accomplish this through *global includes* functionality. See "Oracle JSP Global Includes" on page 7-6.

- Specify additional package names or fully qualified class names through the JSP `extra_imports` configuration parameter, or by using the `ojspc -extraImports` option for pretranslation. Syntax varies between OC4J configuration parameter settings and `ojspc` option settings, so refer to the following sections as appropriate:
  - "JSP Configuration Parameter Descriptions" on page 3-13
  - "Option Descriptions for ojspc" on page 7-13

JSP File Naming Conventions

The file name extension `.jsp` for JSP pages is required by the servlet specification. The servlet 2.3 specification does not, however, distinguish between complete pages that are independently translatable and page segments that are not (such as files brought in through an `include` directive).

The JSP 1.2 specification recommends the following:

- Use the `.jsp` extension for top-level pages, dynamically included pages, and pages that are forwarded to—pages that are translatable on their own.

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

Removal of tools.jar from OC4J Standalone

The OC4J 9.0.3 standalone implementation provided the `tools.jar` file from the Sun Microsystems JDK 1.3.1. This file includes the `java` front-end executable and `javac` compiler executable, for example, among many other components.

The OC4J 10.1.2 standalone implementation no longer provides the `tools.jar` file. Therefore, you must install a JDK that OC4J supports before installing OC4J itself. The JDK versions that OC4J supports for the OC4J 10.1.2 implementation are JDK 1.3.1 (for OC4J standalone only) and JDK 1.4. Oracle Application Server 10*g* Release 2 (10.1.2) includes JDK 1.4, so you should typically use this JDK version for OC4J standalone as well. However, there are migration issues to consider, particularly the JDK 1.4 requirement that all invoked classes must be in packages. See "JDK 1.4 Considerations: Cannot Invoke Classes Not in Packages" below.

Notes: OC4J standalone uses `javac` from the same directory in which `java` is accessed through the command "`java -jar oc4j.jar`", ensuring use of the appropriate `javac` version.

JDK 1.4 Considerations: Cannot Invoke Classes Not in Packages

Among the migration considerations in moving to a Sun Microsystems JDK 1.4 environment, which is the environment that is shipped with Oracle Application Server 10*g* Release 2 (10.1.2), there is one of particular importance to servlet and JSP developers.

As stated by Sun Microsystems, "The compiler now rejects import statements that import a type from the unnamed namespace." This was to address security concerns and ambiguities with previous JDK versions. Essentially, this means that you cannot invoke a class (a method of a class) that is not within a package. Any attempt to do so will result in a fatal error at compilation time.

This especially affects JSP developers who invoke JavaBeans from their JSP pages, as such beans are often outside of any package (although the JSP 2.0 specification now requires beans to be within packages, in order to satisfy the new compiler requirements). Where JavaBeans outside of packages are invoked, JSP applications that were built and executed in an OC4J 9.0.3 / JDK 1.3.1 environment will no longer work in an OC4J 10.1.2 / JDK 1.4 environment.

Until you update your application so that all JavaBeans and other invoked classes are within packages, you can avoid this issue by reverting back to a JDK 1.3.1 environment for OC4J standalone. Note that JDK 1.3.x is not supported in a full Oracle Application Server 10.1.2 environment.

Notes:

- The `javac -source` compiler option is intended to allow JDK 1.3.1 code to be processed seamlessly by the JDK 1.4 compiler, but this option does not account for the "classes not in packages" issue.
- Only the JDK 1.3.1 and JDK 1.4 compilers are supported and certified by OC4J. It is possible to specify an alternative compiler by adding a `<java-compiler>` element to the `server.xml` file, and this might provide a workaround for the "classes not in packages" issue, but no other compilers are certified or supported by Oracle for use with OC4J. (Furthermore, do *not* update the `server.xml` file directly in an Oracle Application Server environment. Use the Oracle Enterprise Manager 10*g*.)

For more information about the "classes not in packages" issue and other JDK 1.4 compatibility issues, refer to the following Web site:

`http://java.sun.com/j2se/1.4/compatibility.html`

In particular, click the link "Incompatibilities Between Java 2 Platform, Standard Edition, v1.4.0 and v1.3".

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.

- `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`: for XML functionality on the client
- `ojdbc14.jar`: for the 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 Oracle Application Server Java Object Cache (which is the default back-end repository for the OC4J Web Object Cache)

JSP Configuration in OC4J

The following sections cover 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 10*g* in an Oracle Application Server environment, such as for production deployment, see "JSP Configuration in Oracle Enterprise Manager 10g" on page 3-23.
- For non-OC4J environments, use the old `oracle.jsp.JspServlet` front-end servlet instead of the `oracle.jsp.runtimev2.JspServlet` version.

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>

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

See the *Oracle Application Server 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. The following subsections provide a summary table, detailed 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
- JSP Configuration Parameter Descriptions
- Setting JSP Configuration Parameters in OC4J

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 pretranslating, and whether the parameter is for runtime or compile-time use.

Notes: See "Option Descriptions for ojspc" on page 7-13 for information about any `ojspc` options.

Table 3–1 JSP Configuration Parameters, OC4J Environment

Parameter	Related ojspc Option	Description	Default	Runtime / Compile-Time
check_page_scope	(None)	Set this boolean to `true` to enable page-scope checking by `JspScopeListener` (OC4J only).	`false`	Runtime
debug_mode	(None)	Set this boolean to `true` to print the stack trace when a runtime exception occurs.	`false`	Runtime
emit_debuginfo	(None)	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 an alternative Java compiler. If you use this option, the compiler will be run in a separate process from OC4J. The `javaccmd` parameter is ignored if `use_old_compiler` is set to `false`.	`null`	Compile-time

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

Parameter	Related ojspc Option	Description	Default	Runtime / Compile-Time
main_mode	(None)	This determines whether JSP-generated classes are automatically reloaded or JSP pages are automatically retranslated, 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	(None)	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
setproperty_onerr_continue	(None)	Set this boolean to `true` to continue iterating over request parameters and setting corresponding bean properties when an error is encountered during `jsp:setProperty` when `property="*"`.	false	Runtime
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	-tagReuse	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

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

Parameter	Related ojspc Option	Description	Default	Runtime / Compile-Time
use_old_compiler	(None)	Set this boolean to `false` to force the JSP container to use the same compiler as the rest of OC4J. Otherwise, by default, OC4J uses in-process compilation (or compilation according to the `javaccmd` setting, if applicable).	`true`, if `tools.jar` in classpath	Compile-time
well_known_taglib_loc	(None)	If TLD caching is *not* enabled, 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, where the Oracle-specific implementation is not used and you must use the `checkPageScope` custom tag for `JspScopeListener` page-scope functionality. See the *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference* for detailed information about the `JspScopeListener` utility.

debug_mode (boolean; default: `false`)

Use the `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.

Note: Oracle JDeveloper enables `emit_debuginfo`.

---

external_resource (boolean; default: `false`)

Set this flag to `true` to instruct the JSP translator to place static content of the page 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 Oracle Application Server 10*g* Release 2 (10.1.2), 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 or Significant Tag Library Usage" on page 6-6.

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

---

extra_imports (import list; default: `null`)

As described in "Default Package Imports" on page 3-7, the default list of packages that are imported into each JSP page is smaller than the list prior to the OC4J 9.0.3 implementation. 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 3-19 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>
```

Notes:

- For pretranslating 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 7-6.

---

forgive_dup_dir_attr (boolean; default: `false`)

Set this boolean to `true` to avoid translation errors in a JSP 1.2 (or higher) environment 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 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 6-8 for more information.

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

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

---

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

If `use_old_compiler` is set to `true`, you can use `javaccmd`, typically during development, to specify a Java compiler command line for use during JSP translation. This would be useful if you want to specify particular `javac` settings or an alternative compiler (optionally including command-line settings). 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.

Be aware of the following:

- The `javaccmd` is ignored if `use_old_compiler` is set to `false`.
- Using `javaccmd` results in the compiler running in a separate process from OC4J.

See "JSP Compilation Considerations" on page 3-4 for related information.

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 pretranslating pages, the `ojspc -noCompile` option allows similar functionality. It results in no compilation by `javac`, so you can compile the translated classes manually through any desired compiler.

---

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

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

Here are the supported settings:

- `justrun`: The runtime dispatcher will not perform any timestamp checking, so there is no retranslation of JSP pages or reloading of JSP-generated Java classes. This mode is the most efficient mode for a deployment environment, where code will not change.
- `reload`: The dispatcher will check the timestamp of classes generated by the JSP translator, such as page implementation classes, and reload any that have changed

or been redeployed since they were last loaded. This might be useful, for example, when you deploy or redeploy compiled classes, but not page source, from a development environment to a production environment.

- `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 `reload` functionality as well.

no_tld_xml_validate (boolean; 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 8-6 for related information.

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

old_include_from_top (boolean; default: `false`)

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

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

precompile_check (boolean; 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 pretranslated 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 Pretranslation without Execution" on page 7-27, and the Sun Microsystems *JavaServer Pages Specification*.

reduce_tag_code (boolean; 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 8-32.

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

req_time_introspection (boolean; 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, however, there is no request-time introspection regardless of the setting of this flag.

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 8-32 for information about use of `VariableInfo`.)

An additional effect of this flag is to allow a bean to be declared twice, such as in different branches of an `if..then..else` loop. Consider the example that follows. With the default `false` value of `req_time_introspection`, this code would cause a parse exception. With a `true` value, the code will work without error:

```
<% if (cond) { %>
      <jsp:useBean id="foo" class="pkgA.Foo1" />
<% } else { %>
      <jsp:useBean id="foo" class="pkgA.Foo2" />
<% } %>
```

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

setproperty_onerr_continue (boolean; default: `false`)

Set this boolean to `true` to continue iterating over request parameters and setting corresponding bean properties when an error is encountered during a `jsp:setProperty` statement when `property="*"`.

See the description of `jsp:setProperty`, under "Standard Actions: JSP Tags" on page 1-12, for related information.

static_text_in_chars (boolean; 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 9-4 for related information.)

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

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

tags_reuse_default (mode 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.

Notes:

- If you use a value of `runtime`, and your code allows the JSP container to continue processing a JSP page in the event that custom tags cause exceptions, you may encounter subsequent occurrences of `ClassCastException`. In this event, change the `tags_reuse_default` value to `compiletime` or `compiletime_with_release`.
- 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 retranslate the JSP pages.
- 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`.
- For pretranslating pages, the `ojspc -tagReuse` option is equivalent.

See "Disabling or Enabling Runtime or Compile-Time Tag Handler Reuse" on page 8-30 for more information about tag handler reuse.

use_old_compiler (boolean; default: `true` if `tools.jar` is in classpath)

You can set `use_old_compiler` to `false` to force the JSP container to use the same compiler as the rest of OC4J—out-of-process compilation with `javac` by default, or compilation according to a `<java-compiler>` element in `server.xml`. The `use_old_compiler` flag is set to `true` by default if `tools.jar` is in the classpath, resulting in in-process compilation unless `javaccmd` is set. (You can use a `<library>` element in the `server.xml` file to ensure that `tools.jar` is in the classpath.)

See "JSP Compilation Considerations" on page 3-4 for related information.

Notes:

- If `tools.jar` is not in the classpath, then `use_old_compiler` is forced to a `false` setting.
- If you want to use an out-of-process compiler, but not the compiler that the rest of OC4J uses, then set `use_old_compiler` to `true` and use the `javaccmd` parameter to specify the desired compiler.

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

If persistent TLD caching is *not* enabled, you can use `well_known_taglib_loc` to specify a single directory to use as the "well-known location" where tag library JAR files can be placed for sharing across multiple Web applications. See "TLD Caching and Well-Known Tag Library Locations" on page 8-16 for important related information.

Specify a relative directory location. This would be 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 is as follows:

- *`ORACLE_HOME`*`/j2ee/home/jsp/lib/taglib/` if *`ORACLE_HOME`* is defined.

or:

- `./jsp/lib/taglib` if *`ORACLE_HOME`* is not defined.

xml_validate (boolean; 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 pretranslating 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 directly 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 3-10, the settings would go where the *`init_params`* placeholder appears.

Note: In an Oracle Application Server production environment, use Enterprise Manager for configuration. You can use the Application Server Control Console Web Module Advanced Properties Page in Enterprise Manager to update the `global-web-application.xml` or `orion-web.xml` file. This Application Server Control Console is discussed in the *Oracle Application Server Containers for J2EE Servlet Developer's Guide*.

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 *Oracle Application Server 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 *Oracle Application Server 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.
- `jsp-cache-directory`: The JSP cache directory is used as a base directory for output files from the JSP translator. (See "JSP Translator Output File Locations" on page 7-5.) It is also used as a base directory for application-level TLD caching. (See "TLD Cache Features and Files" on page 8-17.) The default value is "`./persistence`", relative to the deployment directory of the application.
- `jsp-cache-tlds`: This flag indicates whether persistent TLD caching is enabled for JSP pages. TLD caching is implemented both at a global level, for TLD files in "well-known" tag library locations, and at an application level, for TLD files under the `WEB-INF` directory. Use a "`true`" or "`on`" setting, which is the default, to search

for TLD files among all application files. A setting of "`standard`" searches for TLD files only in `/WEB-INF` and subdirectories other than `/WEB-INF/classes` or `/WEB-INF/lib`. A setting of "`false`" or "`off`" disables this feature. Well-known locations are according to the `jsp-taglib-locations` attribute. See "TLD Caching and Well-Known Tag Library Locations" on page 8-16 for related information.

- `jsp-taglib-locations`: If persistent TLD caching is enabled, through the `jsp-cache-tlds` attribute, you can use `jsp-taglib-locations` to specify a semicolon-delimited list of one or more directories to use as "well-known" locations. Tag library JAR files can be placed in these locations for sharing across multiple Web applications and for TLD caching. See "TLD Caching and Well-Known Tag Library Locations" on page 8-16 for important related information.

  You can specify any combination of absolute directory paths or relative directory paths. Relative paths would be 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 is as follows:

  – *`ORACLE_HOME`*`/j2ee/home/jsp/lib/taglib/` if *`ORACLE_HOME`* is defined.

  or:

  – `./jsp/lib/taglib` if *`ORACLE_HOME`* is not defined.

  Important: Use the `jsp-taglib-locations` attribute only in `global-web-application.xml`, not in `orion-web.xml`.

- `simple-jsp-mapping`: Set this to "`true`" if the "`*.jsp`" file extension is mapped to *only* the `oracle.jsp.runtimev2.JspServlet` front-end JSP servlet in the `<servlet>` elements of any Web descriptors affecting your application (`global-web-application.xml`, `web.xml`, and `orion-web.xml`). This will allow performance improvements for JSP pages. The default setting is "`false`".

- `enable-jsp-dispatcher-shortcut`: A "`true`" setting, which is the case by default, results in significant performance improvements by the OC4J JSP container, especially in conjunction with a "`true`" setting for the `simple-jsp-mapping` attribute. This is particularly true for JSP pages with numerous `jsp:include` statements. Use of the "`true`" setting assumes, however, that if you define JSP files with `<jsp-file>` elements in `web.xml`, then you have corresponding `<url-pattern>` specifications for those files, as in the following example:

  ```
  <servlet>
     <servlet-name>foo</servlet-name>
     <jsp-file>bar.jsp</jsp-file>
  </servlet>
  ...
  <servlet-mapping>
     <servlet-name>foo</servlet-name>
     <url-pattern>/mypath</url-pattern>
  </servlet-mapping>
  ```

  If you use `<jsp-file>` without a corresponding `<url-pattern>` setting (which is *not* a typical scenario), then set `enable-jsp-dispatcher-shortcut="false"`.

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 Oracle Application Server 10*g* Release 2 (10.1.2). You can use the JSP `main_mode` configuration parameter, described in "JSP Configuration Parameter Descriptions" on page 3-13, for functionality equivalent to that of `autoreload-jsp-pages`.

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 *Oracle Application Server Containers for J2EE Servlet Developer's Guide*.

Note: Update these files directly only if you are in an OC4J standalone environment. In an Oracle Application Server environment, use Enterprise Manager for configuration. In Oracle Application Server 10*g* Release 2 (10.1.2), JSP `<orion-web-app>` attributes are not yet supported by the Application Server Control Console JSP Properties Page in Enterprise Manager, but you can make settings through the Application Server Control Console Web Module Advanced Properties Page. This page is described in the *Oracle Application Server Containers for J2EE Servlet Developer's Guide*.

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 the application descriptor of the OC4J default application, which by default is the parent application of any other OC4J applications. Do not confuse this with standard J2EE application-level `application.xml` files. The application descriptor of the OC4J default application is OC4J-specific, governed by `orion-application.dtd`.
- `data-sources.xml`: This specifies data sources for database connections.

(In Oracle Application Server, 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 7-23. For complete information about the use of these files, see the *Oracle Application Server Containers for J2EE User's Guide* and the *Oracle Application Server Containers for J2EE Servlet Developer's Guide*.

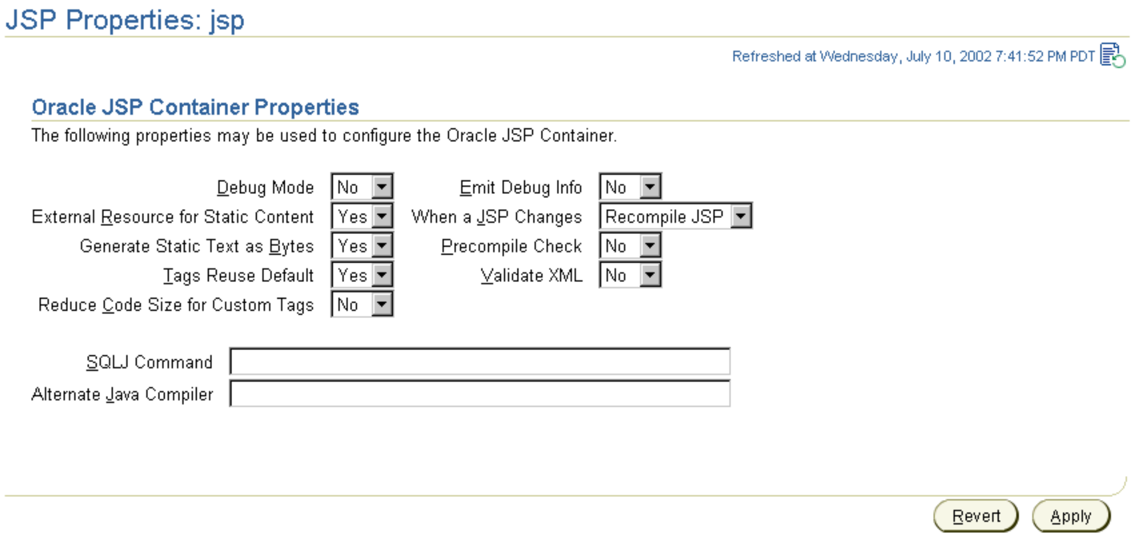
JSP Configuration in Oracle Enterprise Manager 10*g*

In an Oracle Application Server environment, such as for production deployment, use Enterprise Manager for OC4J configuration. This includes configuration of the front-end JSP servlet for the OC4J JSP container.

Oracle Enterprise Manager 10*g* Application Server Control Console is the administration console for an Oracle Application Server instance. It enables you to monitor real-time performance, manage Oracle Application Server components and instances, and configure these components and instances. This includes any instances of OC4J. In particular, Application Server Control Console includes the JSP Properties Page. Application Server Control Console comes with your Oracle Application Server installation. Log in as the `ias_admin` user.

Application Server Control Console JSP Properties Page

The following graphic shows the key portion of the Application Server Control Console JSP Properties Page for an OC4J instance.



JSP Properties: jsp

Refreshed at Wednesday, July 10, 2002 7:41:52 PM PDT

Oracle JSP Container Properties

The following properties may be used to configure the Oracle JSP Container.

Debug Mode	No	Emit Debug Info	No
External Resource for Static Content	Yes	When a JSP Changes	Recompile JSP
Generate Static Text as Bytes	Yes	Precompile Check	No
Tags Reuse Default	Yes	Validate XML	No
Reduce Code Size for Custom Tags	No		

SQLJ Command

Alternate Java Compiler

When you first access an Oracle Application Server instance through Application Server Control Console in Enterprise Manager, you reach the Oracle Application Server Instance Home Page. You can drill down to the JSP Properties Page as follows:

1. From the Oracle Application Server Instance Home Page, select the name of an OC4J instance in the System Components table. Things brings you to the OC4J Home Page for the OC4J instance.
2. From the OC4J Home Page, click **Administration**. This brings you to the OC4J Administration Page.
3. From the OC4J Administration Page, click **JSP Container Properties** under Instance Properties. This brings you to the JSP Properties Page

For further information about using Enterprise Manager, see the *Oracle Application Server Containers for J2EE Servlet Developer's Guide* for an overview of Web application deployment and configuration or the *Oracle Application Server Containers for J2EE User's Guide* for information about any OC4J-related deployment and configuration.

Configuration Parameters Supported by the JSP Properties Page

Table 3–2 shows the correspondence between JSP container properties shown in the Application Server Control Console JSP Properties Page in Enterprise Manager, and configuration parameters of the JSP container front-end servlet as described in "JSP Configuration Parameters" on page 3-11. See that section for the meanings of the settings.

Possible settings are shown with defaults in bold. Note that Application Server Control Console 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 Application Server Control Console Properties, JSP Parameters

Application Server Control Console JSP Container Property	Possible Settings	JSP Configuration Parameter	Possible Settings
Debug Mode	**No**	debug_mode	**false**
	Yes		true

Table 3–2 (Cont.) Application Server Control Console Properties, JSP Parameters

Application Server Control Console JSP Container Property	Possible Settings	JSP Configuration Parameter	Possible Settings
External Resource for Static Content	No	external_resource	**false**
	Yes		true
Generate Static Text as Bytes	No	static_text_in_chars	**false**
	Yes		true
Tags Reuse Default	No	tags_reuse_default	none
	Yes		**runtime**
Reduce Code Size for Custom Tags	**No**	reduce_tag_code	**false**
	Yes		true
Emit Debug Info	**No**	emit_debuginfo	**false**
	Yes		true
When a JSP Page Changes	**Recompile JSP**	main_mode	**recompile**
	Reload Classes		reload
	Do Nothing		justrun
Precompile Check	**No**	precompile_check	**false**
	Yes		true
Validate XML	**No**	xml_validate	**false**
	Yes		true
Alternate Java Compiler	Command string (null by default)	javaccmd	Command string (null by default)

Notes:

- As of Oracle Application Server 10*g* Release 2 (10.1.2), Application Server Control Console 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 Application Server Control Console.
- The Application Server Control Console 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

As of Oracle Application Server 10*g* Release 2 (10.1.2), the following configuration parameters are not yet supported through the Application Server Control Console 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`, `jsp-cache-directory`, `jsp-cache-tlds`, `jsp-taglib-locations`, `simple-jsp-mapping`, and `enable-jsp-dispatcher-shortcut`.

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 Application Server Control Console Web Module Advanced Properties Page, as described in the *Oracle Application Server Containers for J2EE Servlet Developer's Guide*. Also see "Setting JSP Configuration Parameters in OC4J" on page 3-19 and "Setting JSP-Related OC4J Configuration Parameters" on page 3-22 for related information.

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 sections are included:

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 6-12.

Therefore, it is sometimes necessary to go back and forth between servlets and JSP pages in an application. The following sections discuss how to accomplish this:

Important: This discussion assumes a servlet 2.2 or higher environment, such as OC4J (servlet 2.3).

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 1-12.) 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" />
```

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 1-12.

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 4-3.

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

The following sections discuss OC4J JSP and Oracle features to consider when accessing data:

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

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 instance from within another Oracle Database instance; 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. It is assumed that you are already at least somewhat familiar with JDBC basics, but you can refer to the *Oracle Database JDBC Developer's Guide and Reference*.

The OC4J JSP container also supports EJB calls.

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 *Oracle Application Server 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:

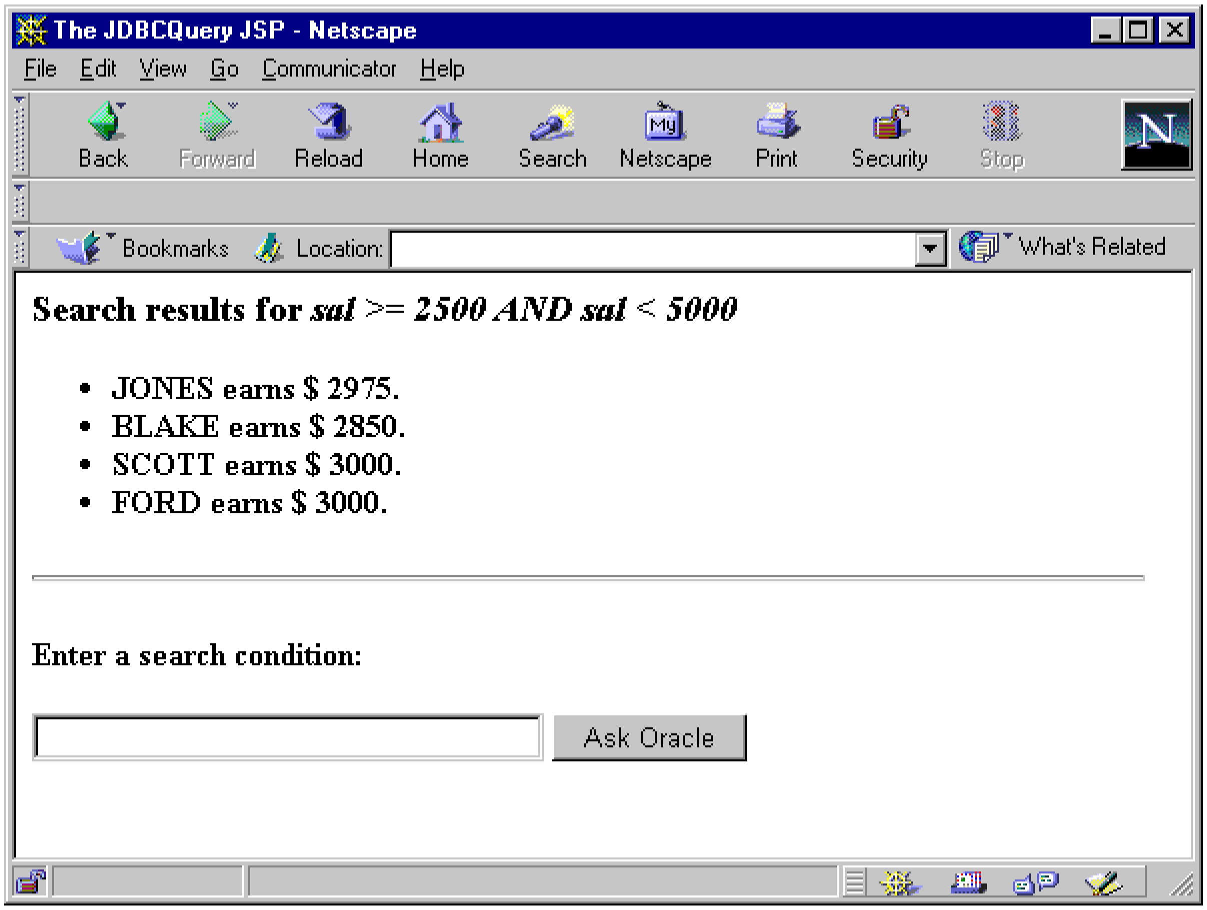
- Another approach to this example would be to define the `runQuery()` method in `<%...%>` scriptlet syntax instead of `<%!...%>` declaration syntax.
- This example uses the JDBC OCI driver, which requires an Oracle client installation. If you want to run this sample, use an appropriate JDBC driver and 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:oci:@",
                                           "scott", "tiger");
        stmt = conn.createStatement();
        // dynamic query
        rset = stmt.executeQuery ("SELECT ename, sal FROM scott.emp "+
                          (cond.equals("") ? "" : "WHERE " + cond ));
       return (formatResult(rset));
     } catch (SQLException e) {
         return ("<P> SQL error: <PRE> " + e + " </PRE> </P>\n");
     } finally {
         if (rset!= null) rset.close();
         if (stmt!= null) stmt.close();
```

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

The graphic below illustrates sample output for the following input:

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


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

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

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

Enter a search condition:

Use of JDBC Performance Enhancement Features

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

- 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 *Oracle Application Server 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 *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference*.

For information about the Oracle JDBC connection caching scheme, see the *Oracle Database 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 *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference*.

For information about the Oracle JDBC statement caching scheme, see the *Oracle Database 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 *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference*.

For more information about Oracle JDBC update batching, see the *Oracle Database 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 *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference*.

For more information about Oracle JDBC row prefetching, see the *Oracle Database 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 application server.

In Oracle Database, the Oracle JDBC implementation 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 *Oracle Database 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. The following sections provide an overview of EJB usage:

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

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 *Oracle Application Server 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. 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. This is 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 to 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 *Oracle Application Server 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 and the library support classes (tag handler classes and tag-extra-info classes) must be made accessible to your application.

OracleXMLQuery Class

The `oracle.xml.sql.query.OracleXMLQuery` class is part of the Oracle Database XML-SQL utility for XML functionality in database queries. This class requires file `xsu12.jar`, which is also required for XML functionality in some of the custom tags and JavaBeans provided with OC4J. This file is provided with Oracle Database and Oracle Application Server.

For information about the `OracleXMLQuery` class and other XML-SQL utility features, refer to the *Oracle XML Developer's Kit Programmer's Guide*.

JSP Resource Management

The following sections discuss standard features and Oracle value-added features for resource management:

- Standard Session Resource Management: HttpSessionBindingListener
- Overview of Oracle Value-Added Features for Resource Management

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 4-5 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 reaches a timeout 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 "UseJDBCQueryBean JSP Page" on page 4-12), 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.

UseJDBCQueryBean JSP Page

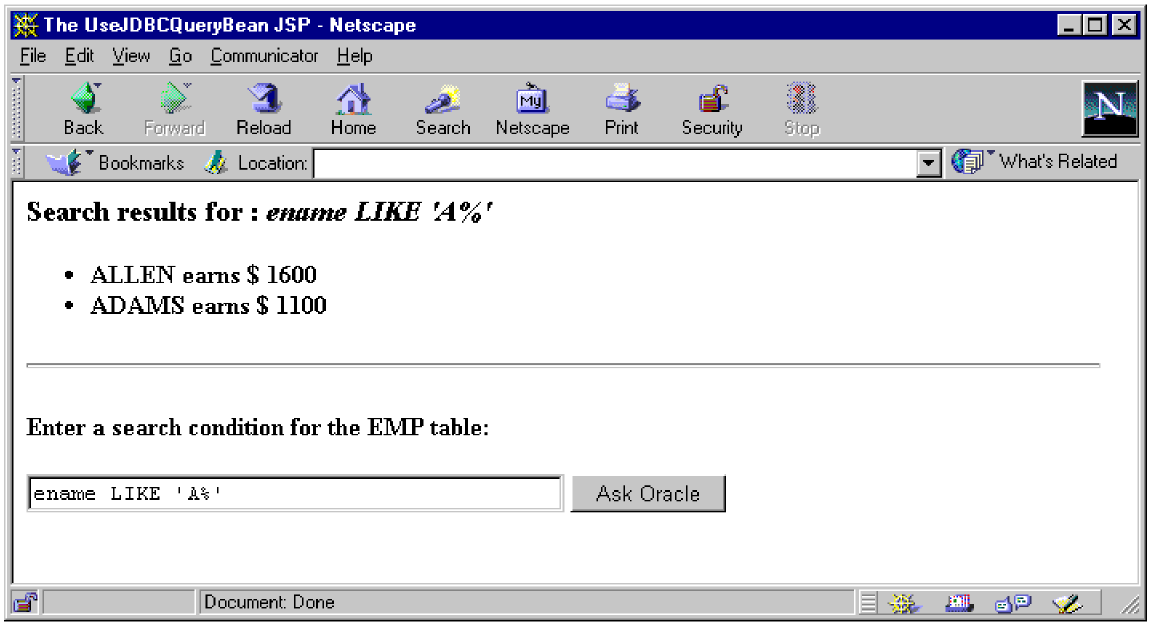
The following JSP page uses the `JDBCQueryBean` JavaBean defined in "JDBCQueryBean JavaBean Code" above, invoking the bean with `session` scope. It uses `JDBCQueryBean` to display employee names that match a search condition entered by the user.

`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:


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

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

- ALLEN earns $ 1600
- ADAMS earns $ 1100

Enter a search condition for the EMP table:

ename LIKE 'A%' | Ask Oracle

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 `JspScopeListener` interface 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 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 *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference.*

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 1-6.)

In a servlet 2.2 or higher environment, you can also specify a default error page in the `web.xml` deployment descriptor through 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 `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 1-10.

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 might 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.

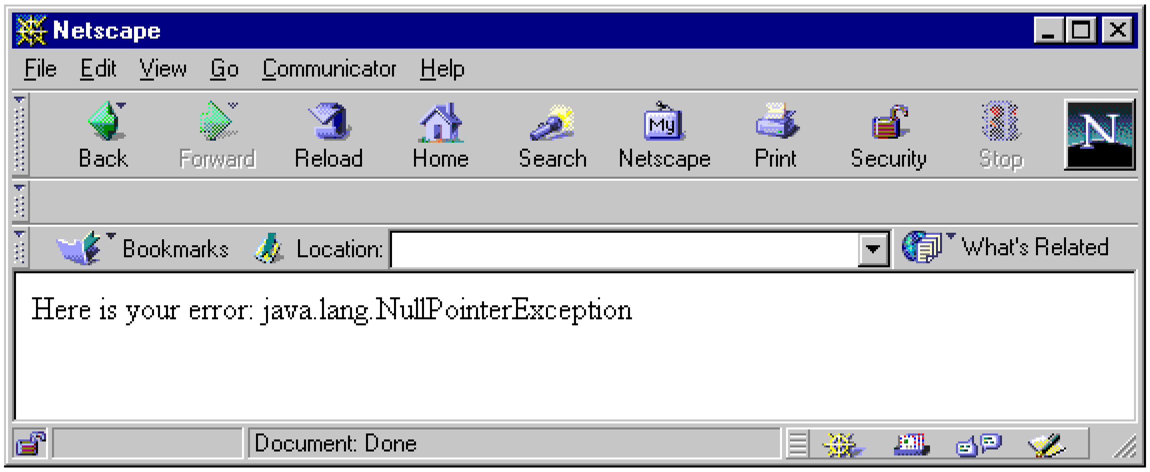
Code for nullpointer.jsp

```
<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:


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

Here is your error: java.lang.NullPointerException

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

5 JSP XML Support

Because of additional support for XML introduced 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 sections:

For information about additional JSP support for XML and XSL, furnished in OC4J through custom tags, refer to the *Oracle Application Server 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. This issue was first addressed in the JSP 1.1 specification by defining equivalent XML-compatible syntax. In JSP 1.1, however, support for this syntax by a JSP container was optional.

Beginning with the JSP 1.2 specification, there is more complete support for XML-compatible JSP syntax, adding features and requiring support by compliant JSP containers.

Note: Prior to Oracle9*i*AS Release 2 (9.0.3), the OC4J JSP container 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 current JSP 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 no longer true.

The term *JSP XML document* (called *JSP document* in the JSP 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 5-3 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 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 5-11 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*.

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 5-5 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

The following subsection 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 5-5.
	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 5-6.	`<%@ page ... %>`
JSP `include` directive element: `<jsp:directive.include ... />` See "JSP XML Directive Elements" on page 5-6.	`<%@ include ... %>`

JSP declaration element:
`<jsp:declaration>`
`   `*`declaration`*
`</jsp:declaration>`
 	
See "JSP XML Declaration, Expression, and Scriptlet Elements" on page 5-7.
	`<%!` *`declaration`* `%>`

JSP expression element:
`<jsp:expression>`
`   `*`expression`*
`</jsp:expression>`
 	
See "JSP XML Declaration, Expression, and Scriptlet Elements" on page 5-7.
	`<%=` *`expression`* `%>`

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

JSP XML Syntax	Corresponding JSP Traditional Syntax
JSP scriptlet element: `<jsp:scriptlet>` *`code fragment`* `</jsp:scriptlet>` See "JSP XML Declaration, Expression, and Scriptlet Elements" on page 5-7.	`<%` *`code fragment`* `%>`
JSP standard action, such as `jsp:include` or `jsp:forward` See "JSP XML Standard Action and Custom Action Elements" on page 5-8.	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 5-8.	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 5-8.	`<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 5-8.	Template data
Other XML elements. These can appear anywhere a `<jsp:text>` element can appear. See "JSP XML Text Elements and Other Elements" on page 5-8.	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 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 Sun Microsystems *JavaServer Pages Specification, Version 1.2* 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 8-11.

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 8-14 and "Packaging and Accessing Multiple Tag Libraries in a JAR File" on page 8-13.

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

Consider the following `page` directive:

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

This is equivalent to the following JSP XML element:

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

Example: include Directive

Consider the following `include` directive:

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

This 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

Consider the following JSP declaration:

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

This 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 "`&lt;`" escape character instead of "<":

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

Example: JSP Expression

Consider the following JSP expression:

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

This is equivalent to the following JSP XML element:

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

Example: JSP Scriptlet

Consider the following JSP scriptlet:

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

This 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 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://www.oracle.com/technology/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 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 8-36.) 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.

These topics are covered in the following sections:

- 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* 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 5-12.

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 5-5.
- 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 5-5.
- It converts each `page` directive into the equivalent element in JSP XML syntax. See "JSP XML Directive Elements" on page 5-6.
- 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 5-7.
- It converts request-time expressions into XML syntax. See "JSP XML Standard Action and Custom Action Elements" on page 5-8.

- It creates `<jsp:text>` elements for template data. See "JSP XML Text Elements and Other Elements" on page 5-8.
- 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 5-8, 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 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 8-36 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, followed by the XML view of the page as generated by the OC4J JSP translator. The code displays the Oracle JSP version number and configuration parameter values.

Traditional JSP Page

Here is the traditional JSP page:

```
<HTML>
    <HEAD>
        <TITLE>JSP Information </TITLE>
    </HEAD>
    <BODY BGCOLOR="#FFFFFF">
         JSP Version:<BR>
            <%= application.getAttribute("oracle.jsp.versionNumber") %>
         <BR>
         JSP 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>JSP Information </TITLE>
       </HEAD>
       <BODY BGCOLOR="#FFFFFF">
            JSP Version:<BR>]]></jsp:text>
   <jsp:expression jsp:id="2">
      <![CDATA[ application.getAttribute("oracle.jsp.versionNumber") ]]>
   </jsp:expression>
   <jsp:text jsp:id="3"><![CDATA[
            <BR>
            JSP 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 Programming Considerations

This chapter discusses an assortment of programming strategies and tips for use in developing JSP applications. The following sections are included:

- General JSP Programming Strategies
- Additional JSP Programming Tips

General JSP Programming Strategies

This portion discusses issues you should consider when programming JSP pages, regardless of the particular target environment. The following sections are included:

- JavaBeans Versus Scriptlets
- Static Includes Versus Dynamic Includes
- When to Consider Creating and Using JSP Tag Libraries

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 1-4 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 4-5, 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 1-6, 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 1-12, dynamically includes output from the included page within the output of the including page during execution. 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. It is as though the text of the included page is physically copied into the including page, at the point of the `include` directive, during translation. 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 of Dynamic and Static Includes

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 or Significant Tag Library Usage" on page 6-6.)

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 6-5.)

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 7-6.

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 JSP developers might not be experienced in Java programming, they might 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.

Two examples of this are the JavaServer Pages Standard Tag Library (JSTL), supported by OC4J, and the JSP Markup Language (JML) tag library that is provided with OC4J. These libraries are discussed in the *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Providing Convenient JSP Programming Access to API Features

Instead of having Web application programmers rely on Java APIs for using product functionality or extensions from servlets or JSP scriptlets , 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.

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>
```

Additional JSP Programming Tips

In addition to the general programming strategies described earlier, there are a variety of programming tips to consider, as described in the following sections:

- Hiding JSP Pages from Direct Invocation
- Use of a Central Checker Page
- Workarounds for Large Static Content or Significant Tag Library Usage
- Method Variable Declarations Versus Member Variable Declarations
- Page Directive Characteristics
- JSP Preservation of White Space and Use with Binary Data

Hiding JSP Pages from Direct Invocation

There are situations, particularly in an architecture such as Model-View-Controller (MVC), where you would want to ensure that some JSP pages are accessible only to the application itself and cannot be invoked directly by users.

As an example, assume that the front-end or "view" page is `index.jsp`. The user starts the application through a URL request that goes directly to that page. Further assume that `index.jsp` includes a second page, `included.jsp`, and forwards to a third page, `forwarded.jsp`, and that you do not want users to be able to invoke these directly through a URL request.

A mechanism for this is to place `included.jsp` and `forwarded.jsp` in the application `/WEB-INF` directory. When located there, the pages cannot be directly invoked through URL request. Any attempt would result in an error report from the browser.

The page `index.jsp` would have the following statements:

```
<jsp:include page="WEB-INF/included.jsp"/>
...
<jsp:forward page="WEB-INF/forwarded.jsp"/>
```

The application structure would be as follows, including the standard `classes` directory for any servlets, JavaBeans, or other classes, and including the standard `lib` directory for any JAR files:

```
index.jsp
```

```
WEB-INF/
   web.xml
   included.jsp
   forwarded.jsp
   classes/
   lib/
```

Use of a Central Checker Page

For general management or monitoring of your JSP application, it might 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 are many more possible 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 4-10.)

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

   valueBound(HttpSessionBindingEvent event)
   {...}

   valueUnbound(HttpSessionBindingEvent event)
   {...}

   ...
}
```

You can create a checker page, suppose `centralcheck.jsp`, that contains 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.

Notes:

- OC4J offers *global includes* as a convenient way to statically include a file into multiple pages. See "Oracle JSP Global Includes" on page 7-6.
- Alternatively, you might consider servlet filters for this sort of functionality. Servlet filters are described in the *Oracle Application Server Containers for J2EE Servlet Developer's Guide*.

Workarounds for Large Static Content or Significant Tag Library Usage

JSP pages with large amounts of static content (essentially, large amounts of HTML code without content that changes at runtime) might 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 `jsp:include` tag to include its output in the JSP page output at runtime. See "Standard Actions: JSP Tags" on page 1-12 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 3-13.

  For pretranslation, the `-extres` option of the `ojspc` tool offers equivalent functionality.

  Note: Putting static HTML into a resource file might 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 problem with JSP pages that have large static content, or more commonly with JSP pages that have a great deal of tag library usage, 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 might 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.

If a large amount of tag library usage results in a size limit problem for a JSP page, a common solution is to break the page into multiple pages and use `jsp:include` tags as appropriate.

Method Variable Declarations Versus Member Variable Declarations

In "Scripting Elements" on page 1-7, 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 ... {
   ...

   // ** 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 9-4.

Duplicate Settings of Page Directive Attributes Are Disallowed

The JSP 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 Parameter Descriptions" on page 3-13 for information about this parameter. 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 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 6-1 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="java.io.*, java.sql.*" %>
```

or:

```
<%@ page language="java" %>
<%@ page import="java.io.*" %>
<%@ 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 might 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.

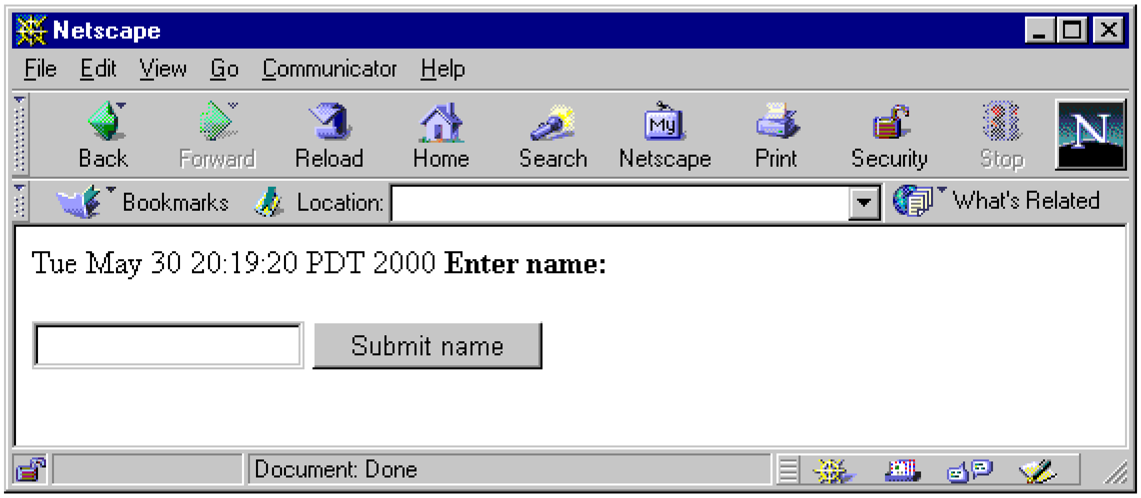
`whitesp.jsp`:

```
<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.


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

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

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 general example:

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

  In this case, the browser will receive an unwanted newline character 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.

  Note: The preceding example is for illustrative purposes only and might not be portable to future Oracle JSP versions or other JSP containers.

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.

7

JSP Translation and Deployment

This chapter discusses operation of the OC4J JSP translator, then discusses the `ojspc` utility and situations where pretranslation is useful, followed by general discussion of a number of additional JSP deployment considerations.

The following sections are included:

- Functionality of the JSP Translator
- The ojspc Pretranslation Utility
- JSP Deployment Considerations

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.

The following sections discuss general functionality of the JSP translator, focusing on its behavior in on-demand translation scenarios such as in OC4J in the Oracle Application Server:

- Features of Generated Code
- General Conventions for Output Names
- Generated Package and Class Names
- Generated Files and Locations
- Issues in the Current Release
- 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` 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 pretranslation model, generated code automatically includes the following features:

- It extends a wrapper class provided by the JSP container that implements the `javax.servlet.jsp.HttpJspPage` interface, which extends the more generic `javax.servlet.jsp.JspPage` interface, which in turn extends the `javax.servlet.Servlet` interface.
- It implements the `_jspService()` method specified by the `HttpJspPage` interface. This method, often referred to 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".

Member Variables 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 places the static text itself in a series of member variables in the page implementation class. The service method `out.print()` statements reference these member variables to print the text.

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 or Significant Tag Library Usage" on page 6-6. You can request this feature through the JSP `external_resource` configuration parameter for on-demand translation, or the `ojspc -extres` flag for pretranslation.

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 Oracle Application Server 10*g* Release 2 (10.1.2), 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 7-4.

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 JSP specification 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 7-2.

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 document 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 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 users specify in the URL must match the case of the actual `.jsp` 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:

- The file name `UserLogin.jsp` results in the class `_UserLogin`.
- The file name `Userlogin.jsp` results in the class `_Userlogin`.
- The file name `userlogin.jsp` results in the class `_userlogin`.

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

Generated Files and Locations

For on-demand translation scenarios, this section describes files that are generated by the JSP translator and where they are placed. (For pretranslation 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 7-22.)

Wherever JSP configuration parameters are mentioned, see "JSP Configuration Parameters" on page 3-11 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 7-2.

Files Generated by the JSP Translator

This section lists files that are generated by the JSP translator. For the file name examples, presume a file `Foo.jsp` is being translated.

Source files:

- A `.java` file (for example, `_Foo.java`) is produced for the page implementation class.

Binary files:

- 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.
- 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 might include variations beyond that.

JSP Translator Output File Locations

The JSP translator places generated output files under a `_pages` directory that is created under the JSP cache directory, which is specified in the `jsp-cache-directory` attribute of the `<orion-web-app>` element in either the `global-web-application.xml` file or the application `orion-web.xml` file. Here is the general base location if you assume the default "`./persistence`" value of `jsp-cache-directory`:

```
ORACLE_HOME/j2ee/home/app-deployment/app-name/web-app-name/persistence/_pages/...
```

In OC4J standalone, here is the location relative to where OC4J is installed:

```
j2ee/home/app-deployment/app-name/web-app-name/persistence/_pages/...
```

Note the following, and refer to "Key OC4J Configuration Files" on page 3-22 for related information about the noted configuration files:

- The *`app-deployment`* directory is the OC4J deployment directory specified in the OC4J `server.xml` file. (In OC4J standalone, this 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 Web site XML file (typically `default-web-site.xml` file in Oracle Application Server or `http-web-site.xml` in OC4J standalone).

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

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

```
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` and `_welcome.class`) in the following directory:

```
j2ee/home/application-deployments/default/defaultWebApp/persistence/_pages/_examples/_jsp
```

Because the `.jsp` source file is in the `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.

Note the following for OC4J standalone:

- 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.

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.

Issues in the Current Release

In conditions where OC4J is heavily loaded and running out of resources, the JSP translator may occasionally produce a zero-length `.class` file, resulting in a "500 Internal Server Error". Use one of the following techniques to remedy the problem:

- Touch the appropriate JSP file so that it will be retranslated and recompiled.
- Remove the zero-length class file. (Its location will be noted in the error output.)

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

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.

Subelement of `<ojsp-global-include>`:

`<include>`

<include ... >

Use the `<include>` 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.

Subelement of `<include>`:

`<into>`

Attributes of `<include>`:

- `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 application-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 of `<into>`:

- `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" %>
```

The ojspc Pretranslation Utility

The `ojspc` utility is provided with OC4J for pretranslation of JSP pages. For consideration of pretranslation scenarios, see "JSP Pretranslation" on page 7-26 and "Deployment of Binary Files Only" on page 7-28.

The following sections discuss `ojspc` functionality:

- Overview of Basic ojspc Functionality
- Overview of ojspc Batch Pretranslation
- Option Summary Table for ojspc
- Command-Line Syntax for ojspc
- Option Descriptions for ojspc

- Summary of ojspc Output Files, Locations, and Related Options

---

Important:

- When you pretranslate JSP pages, be aware that significant differences between the development system and the runtime system, such as a different operating system or JDK, could cause problems.
- For use of `ojspc`, the `tools.jar` file from your JDK must be in the classpath. This is taken care of automatically in an Oracle Application Server installation.
- When using `ojspc` batch pretranslation in the current release, do not place `.java` files in or under the the `/WEB-INF/lib` or `/WEB-INF/classes` directory. Placing `.java` files in or under either of these directories may result in one or more duplicate `.class` files at the top level of the of the archive during batch pretranslation.

Overview of Basic ojspc Functionality

For a 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.
- It invokes the Java compiler to compile the `.java` file, producing a `.class` file for the page implementation class.

Under some circumstances (as is noted in the `-extres` option description later in this chapter), `ojspc` options direct the JSP translator to produce a `.res` Java resource file for static page content, instead of putting this content into 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 7-1.

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

Overview of ojspc Batch Pretranslation

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

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 `.java` files in the archive file.
3. Adds the resulting `.class` files and any Java resource files into a nested JAR file inside the archive file, and discards `.java` files that were created in the process. The `.class` and resource files in the nested JAR file have 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.

By default, the mechanics are that the original archive file is extracted into a temporary storage area, a temporary archive file is created, contents of the original archive file are copied into the temporary file, output `.class` and resource files from pretranslation are added to a nested JAR file within 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. Alternatively, you have the option of specifying a new output file name, which will preserve the original archive file.

The nested JAR file is in the `_pages` directory of the resulting archive file. Its name includes the base name of the resulting archive file (either the original archive file name or a name according to the `ojspc -output` option) and has the `.jar` extension. In the OC4J 10.1.2 implementation, assuming an archive output file name of `myarch.war`, for example, the nested JAR file name would be `__oracle_jsp_myarch.jar`. Implementation details might change in future releases, but the base name of the archive file will always be included in the nested JAR file name.

File paths within the nested JAR file are according to Java package names and according to specified file paths of JSP include and forward statements, as applicable and as one would expect.

Note: The only support for nested archive files is a WAR file within an EAR file. In this circumstance, the contents of the EAR file are extracted, then the contents of the WAR file are extracted and processed and the WAR file is updated appropriately, then the other contents of the EAR file are processed and the EAR file is updated appropriately.

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

- You can use the `-batchMask` option to specify file name extensions for pretranslation and compilation. Whatever you specify is used instead of the defaults (`*.jsp` and `*.java`).
- You can use the `-output` option to specify a new archive file name. By default, `ojspc` updates the original archive file, adding output `.class` files and any resource files (and possibly deleting processed source files, according to the `-deleteSource` option). If you want to be sure that the original archive file is unaltered, then enable the `-output` option. In this case, all contents of the original archive file are copied into the specified output archive file, then the specified output archive file is updated instead of the original 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 processed source files to appear in the resulting archive file. If you do not also use the `-output` option, this effectively means that all processed source files are removed from the original archive file after processing. If you do not use the `-batchMask` option, this consists of all `.jsp` and `.java` files. Otherwise, this consists of all files specified in the `-batchMask` setting.

For examples of these options, see the descriptions of those options under "Option Descriptions for ojspc" on page 7-13.

Option Summary Table for ojspc

Table 7–1 summarizes the options supported by the `ojspc` pretranslation utility. These options are further discussed in "Option Descriptions for ojspc" on page 7-13.

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

Table 7–1 Options for ojspc Pretranslation 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 pretranslation, optionally specify file masks for processing.	`*.jsp`, `*.java`
-dir (or -d)	(None)	Specify the location where `ojspc` should place generated binary files (`.class` and resource). Do not use this option for batch pretranslation.	Current directory
-deleteSource	(None)	For batch pretranslation, use this flag to direct that processed 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 pretranslation.	Empty
-extraImports	extra_imports	Use this to add imports beyond the JSP defaults.	Empty
-extres	external_resource	Use this flag to direct `ojspc` to generate an external resource file for static text from the JSP file.	`false`
-forgiveDupDirAttr	forgive_dup_dir_attr	Use this flag 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)	Use this flag to direct `ojspc` to display usage information.	`false`
-implement	(None)	Specify an interface for the generated page implementation class to implement. Do not use this option for batch pretranslation.	Empty

Table 7–1 (Cont.) Options for ojspc Pretranslation Utility

Option	Related JSP Configuration Parameters	Description	Default
-noCompile	javaccmd	Use this flag to direct `ojspc` to *not* compile the generated page implementation class.	`false`
-noTldXmlValidate	no_tld_xml_validate	Use this flag to *disable* XML validation of TLD files. By default, validation of TLD files is performed.	`false`
-oldIncludeFromTop	old_include_from_top	Use this flag 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 pretranslation, 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 (with package names according to `.jsp` file location)
-reduceTagCode	reduce_tag_code	Use this flag to direct further reduction in the size of generated code for custom tag usage.	`false`
-reqTimeIntrospection	req_time_introspection	Enable this flag in order to allow request-time JavaBean introspection whenever compile-time introspection is not possible.	`false`
-srcdir	(None)	Specify the location where `ojspc` should place generated source files (`.java`). Do not use this option for batch pretranslation.	Current directory
-staticTextInChars	static_text_in_chars	Use this flag to instruct the JSP translator to generate static text in JSP pages as characters instead of bytes.	`false`
-tagReuse	tags_reuse_default	This specifies the mode for JSP tag handler reuse: `runtime` for the runtime model, `compiletime_with_release` or `compiletime` for the compile-time model, or `none` to disable tag handler reuse.	runtime
-verbose	(None)	Use this flag to direct `ojspc` to print status information as it executes.	`false`
-version	(None)	Use this flag to direct `ojspc` to display the JSP version number.	`false`
-xmlValidate	xml_validate	Use this flag to request 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 [option_settings] file_list
```

The file list can include JSP files and other source files (`.java`), 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`.
- Option settings and file names can actually be in any order, including interspersed.

Here are two examples:

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

% ojspc -deleteSource myapp.war
```

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.

-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 7-22.)

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 -appRoot /abc /abc/def/ghi/test.jsp
```

The package is still `_def._ghi`, based on the location of `test.jsp` relative to the specified application root directory.

Note: It is typical for the specified application root directory to be some level of parent directory of the directory where the translated JSP page is located.

-batchMask

(file masks to batch-process in archive; `ojspc` default: see description)

For batch pretranslation, you can use this option to specify source files to process in an archive file. By default, all `.jsp` and `.java` files are processed. File masks specified through the `-batchMask` option are used instead of (not in addition to) these defaults.

Place quotes around the list of file masks and use commas or semicolons as delimiters within the list. White space before or after a file mask is ignored. You can include directories in the mask.

The `-batchMask` implementation includes complete support for standard wildcard pattern-matching.

Given the default setting, the following two examples are equivalent:

```
% ojspc myapp.war

% ojspc -batchMask "*.jsp,*.java" myapp.war
```

This next example drops processing for `.java` files while adding processing for `.jspf` and `.jsph` files:

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

The following example does not process `.java` files, and only processes `.jsp` files whose names start with "abc" and who are in subdirectories under the top level of the archive file:

```
% ojspc -batchMask "*/abc*.jsp" myapp.zip
```

The following example is the same as the preceding example, but also processes `.jsp` files whose names start with "abc" in the top level of the archive file:

```
% ojspc -batchMask "abc*.jsp, */abc*.jsp" myapp.jar
```

This final example specifically processes the file `a.jspc`, as well as any `.jsp` files that start with "My" and are in a directory that is a subdirectory of `mydir/subdir` and matches the pattern "t?st" (any character as the second character, such as "test", "tast", or "tust"):

```
% ojspc -batchMask "mydir/subdir/t?st/My*.jsp, a.jspc" myapp.ear
```

Important: File masks specified in this option are *not* case-sensitive.

-deleteSource

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

For batch pretranslation, enable this flag if you do not want processed source files to appear in the resulting archive file. This is `.jsp` and `.java` files by default, or else only the files that match the file mask in the `-batchMask` option. Generated `.java` files are also discarded, as usual.

If you do not use the `-output` option, then the contents of the original archive file are overwritten to remove any processed source files after processing. If you do use the `-output` option, then processed source files will not be copied to the specified output archive file. (The original archive file is unaltered.)

Notes:

- Files whose names do not match the default file extensions (if you do not use the `-batchMask` option), or whose names do not match the name masks specified using the `-batchMask` option, will not be discarded through the `-deleteSource` option. You must delete these files manually from the resulting archive file if desired. In particular, this applies to statically included source files, which are not translatable on their own and so should not use the `.jsp` extension or any other extension that might result in an attempt to translate the files on their own.
- As in any situation where JSP source files are not deployed, if you use `-deleteSource`, then 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 7-28.

-dir

(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 a shortcut, `-d` is also accepted.

The specified path is taken as a file system path (not an application-relative or page-relative path), and the directory must already exist.

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 7-22 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 `-dir` for batch pretranslation.
- In environments such as Windows that allow spaces in directory names, enclose the directory name in quotes.

-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 pretranslation.

-extraImports

(import list; `ojspc` default: empty)

As described in "Default Package Imports" on page 3-7, the OC4J JSP container has a smaller default list of packages that are imported into each JSP page than was the case prior to Oracle9*i*AS Release 2 (9.0.3). 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 in quotes, and either comma-delimited or semicolon-delimited, as in the following example:

```
% ojspc -extraImports "java.util.*,java.io.*" foo.jsp
```

Notes:

- White space within the quotes, before or after package names or class names, is ignored.
- 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 7-6.

-extres

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

Enabling this flag instructs `ojspc` to place static content of the page 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. 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 output `.class` files.

If there is a lot of static content in a page, this technique will speed translation and might speed execution of the page. For more information, see "Workarounds for Large Static Content or Significant Tag Library Usage" on page 6-6.

Note: 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 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 6-8 for more information.

The JSP 1.1 specification did *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 pretranslation.

-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.

Note: 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.

-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 8-6 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, which complies with the JSP 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 pretranslation, use the `-output` option if you want to specify a new archive file for output instead of updating the original archive file. 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 pretranslation are placed into a nested JAR file within the specified file (and source files are deleted from the specified file if `-deleteSource` is enabled). The original archive file is unaltered and you would use the new file instead of the original file for deployment. (See "Overview of ojspc Batch Pretranslation" on page 7-9 for information about the nested JAR file.)

Without the `-output` option, the original archive file is updated; no new archive file is created.

Here is an example of `-output` usage:

```
% ojspc -output myappout.war myapp.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 -packageName myroot.mypackage src/jspsrc/Foo.jsp
```

This results in `myroot.mypackage` being used as the package name.

If this example did *not* use the `-packageName` option, 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 might be performance consequences regarding tag handler reuse, however. See "Tag Handler Code Generation" on page 8-32.

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, however, there is no request-time introspection regardless of the setting of this flag.

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 `-reqTimeIntrospection` is enabled, the JSP container will delay introspection until request-time. (See "Scripting Variables, Declarations, and Tag-Extra-Info Classes" on page 8-32 for information about use of `VariableInfo`.)

An additional effect of this flag is to allow a bean to be declared twice, such as in different branches of an `if..then..else` loop. Consider the example that follows. Without `-reqTimeIntrospection` being enabled, this code would cause a parse exception. With it enabled, the code will work without error:

```
<% if (cond) { %>
      <jsp:useBean id="foo" class="pkgA.Foo1" />
<% } else { %>
      <jsp:useBean id="foo" class="pkgA.Foo2" />
<% } %>
```

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

-srcdir

(fully qualified path; `ojspc` default: current directory)

Use this option to specify a base directory location for `ojspc` placement of generated source files (`.java` files).

The specified path is taken simply as a file system path, not an application-relative or page-relative path. The directory must already exist.

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 7-22 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 pretranslation.
- In environments such as Windows 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.

-tagReuse

(mode for tag handler reuse; `ojspc` default: `runtime`)

Use this option 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.

Notes:

- If you use a value of `runtime`, and your code allows the JSP container to continue processing a JSP page in the event that custom tags cause exceptions, you may encounter subsequent occurrences of `ClassCastException`. In this event, change the `-tagReuse` value to `compiletime` or `compiletime_with_release`.
- If you switch from the runtime model (`-tagReuse` value of `runtime`) to the compile-time model (`-tagReuse` value of `compiletime` or `compiletime_with_release`), or from the compile-time model to the runtime model, you must retranslate the JSP pages.
- For backward compatibility, a setting of `true` is also supported and is equivalent to a setting of `runtime`, and a setting of `false` is supported and is equivalent to a setting of `none`.
- In an on-demand translation scenario, the JSP `tags_reuse_default` configuration parameter provides the same functionality.

See "Disabling or Enabling Runtime or Compile-Time Tag Handler Reuse" on page 8-30 for more information about tag handler reuse.

-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 -verbose myerror.jsp
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 places them in or under your current directory, from which you ran `ojspc` (not considering batch pretranslation).

Here are the files:

- A `.java` source file (for batch pretranslation, this is discarded after compilation)
- A `.class` file for the page implementation class
- 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 7-4.

To summarize some of the commonly used options described under "Option Descriptions for ojspc" on page 7-13, 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 pretranslation)

- `-dir` to place binary files—`.class` files and Java resource files—in a specified location (not relevant for batch pretranslation)
- `-noCompile` to *not* compile the generated page implementation class source

  As a result of this, no `.class` files are produced.
- `-extres` to put static text into a Java resource file

For output file placement, not considering batch pretranslation, the directory structure underneath the current directory (or directories specified by the `-dir` 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 `-dir` 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 -appRoot /abc /abc/def/ghi/test.jsp
```

The package is still `_def._ghi`, according to the location of `test.jsp` relative to the specified application root. Output files will be placed in the directory `/home/mydir/_def/_ghi` or in a `_def/_ghi` subdirectory under locations specified through the `-dir` and `-srcdir` options. In either case, the `_def/_ghi` subdirectory structure is created as part of the process.

JSP Deployment Considerations

The following sections cover general deployment considerations and scenarios, mostly independent of your target environment:

- Overview of EAR/WAR Deployment
- Application Deployment with Oracle JDeveloper
- JSP Pretranslation
- 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 *Oracle Application Server Containers for J2EE User's Guide* for detailed information about deployment to OC4J in an Oracle Application Server environment.

OC4J Deployment Features

In OC4J, deploy each application through a standard EAR (Enterprise archive) file. The name of the application and the name and location of the EAR file are specified through an `<application>` element in the OC4J `server.xml` file. This file is in the OC4J configuration files directory.

> **Note:** In Oracle Application Server, directory paths are configurable; in OC4J standalone, the configuration files directory is `j2ee/home/config` by default.

In an Oracle Application Server environment, use Enterprise Manager for deployment and configuration and do not update `server.xml` or other configuration files directly. Refer to the *Oracle Application Server Containers for J2EE User's Guide* for information.

In an OC4J standalone development environment, OC4J supports the `admin.jar` tool for deployment. This modifies `server.xml`, `http-web-site.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 Oracle Application Server without going through Enterprise Manager, you must run the `dcmctl` tool, using its `updateConfig` command, to inform Oracle Application Server Distributed Configuration Management (DCM) of the updates. (This does not apply in an OC4J standalone mode, where OC4J is being run apart from Oracle Application Server.)

The `dcmctl` tool is documented in the *Oracle 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

See the *Oracle Application Server Containers for J2EE User's Guide* for more information about deployment to Oracle Application Server. See the *Oracle Application Server Containers for J2EE Stand Alone User's Guide* for deployment to a standalone environment and for information about `admin.jar`. Also see "Key OC4J Configuration Files" on page 3-22 for a summary of important configuration files in OC4J.

Standard WAR Deployment

The JSP specification (since JSP 1.1) supports the packaging and deployment of Web applications, including JavaServer Pages, according to the servlet specification (since servlet 2.2).

In typical 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, 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*.

Note: In OC4J, you typically deploy a WAR file within an EAR file. If you deploy a WAR file directly, OC4J transparently wraps it in an EAR file.

Application Deployment with Oracle JDeveloper

Oracle 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, and business components archive profile.

When creating an Oracle ADF Business Components Web application using Oracle JDeveloper, a J2EE Web module deployment archive is generated, containing both the Oracle ADF Business Components 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 Business Components 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 *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference* for an overview of the Business Components 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 Oracle Application Server instances.

For information about JDeveloper, refer to the JDeveloper online help or to the following site on the Oracle Technology Network:

`http://www.oracle.com/technology/products/jdev/content.html`

JSP Pretranslation

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 pretranslate JSP pages, which offers at least two advantages:

- It saves users the translation overhead the first time a page is invoked.
- It ensures that the developer or deployer, instead of users, will see any translation or compilation errors.

You also might want to pretranslate pages so that you can deploy binary files only, as discussed in "Deployment of Binary Files Only" on page 7-28.

OC4J users can employ the Oracle `ojspc` utility, either specifying individual files for pretranslation or specifying archive files (JAR, WAR, EAR, or ZIP) for batch pretranslation. There is also a standard `jsp_precompile` mechanism. These topics are covered in the following sections:

- Techniques for Page Pretranslation with ojspc
- Batch Pretranslation with ojspc
- Standard JSP Pretranslation without Execution

Also see "The ojspc Pretranslation Utility" on page 7-8 for detailed information about this utility.

Techniques for Page Pretranslation with ojspc

When you pretranslate with `ojspc` (not considering batch pretranslation), use the `-dir` 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 7-5, where the JSP page is located in the `examples/jsp` subdirectory under the OC4J standalone default Web application directory:

```
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 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:

```
j2ee/home/application-deployments/default/defaultWebApp/persistence/_pages/_examples/_jsp
```

When you pretranslate, 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, *`OC4J_HOME`* is the directory where OC4J is installed, and the `ojspc` command wraps around to a second line):

```
% cd OC4J_HOME/j2ee/home/default-web-app
% ojspc examples/jsp/welcome.jsp
-dir OC4J_HOME/j2ee/home/application-deployments/default/defaultWebApp/persistence/_pages
```

(This assumes you specify the appropriate *OC4J_HOME* directory.) The ojspc command translates examples/jsp/welcome.jsp and specifies 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 pretranslated binaries and would not have to perform translation, assuming that either the source file was not altered after pretranslation, 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 Oracle Application Server 10*g* Release 2 (10.1.2).

Batch Pretranslation with ojspc

There are ojspc features for batch pretranslation 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 .java files in the contents will be pretranslated and compiled, as appropriate, and the archive file will be updated to include the output .class files and any Java resource files (but not generated .java files). You would then deploy the resulting archive file.

In addition to this basic functionality, you can use key ojspc options as follows:

- You can use the -batchMask option to specify file name extensions for pretranslation and compilation. Whatever you specify is instead of the defaults (*.jsp and *.java).
- You can use the -output option to specify a new archive file name. By default, ojspc updates the original archive file, adding output .class files and any resource files (and possibly deleting processed source files, according to the -deleteSource option). If you want to be sure that the original archive file is unaltered, then enable the -output option. In this case, all contents of the original archive file are copied into the specified archive file, then the specified file is updated instead of the original 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 processed source files to appear in the resulting archive file. If you do not also use the -output option, this effectively means that all processed source files are removed from the original archive file after processing. If you do not use the -batchMask option, this consists of all .jsp and .java files. Otherwise, this consists of all files specified in the -batchMask setting.

Standard JSP Pretranslation without Execution

It is also possible to specify JSP pretranslation without execution when you invoke the page in an on-demand scenario. Accomplish this as follows:

1. Enable the JSP precompile_check configuration parameter. (See "JSP Configuration Parameters" on page 3-11.)

2. Enable the standard `jsp_precompile` request parameter when you invoke 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* 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 pretranslating the pages and deploying only the translated and compiled binary files. Pages that are pretranslated, 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 pretranslate 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 Pretranslation Utility" on page 7-8 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. Alternatively, 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 will terminate with a "file not found" error if it cannot find the source file.

With `main_mode` set appropriately, the 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 3-19.

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 consists of the following sections:

The chapter offers a detailed overview of standard tag library functionality. For complete information, refer to the Sun Microsystems *JavaServer Pages Specification*. For information about the tag libraries provided with OC4J, see the *Oracle Application Server 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 of the 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 tag library features introduced 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*

- 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 8-5 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 8-11 discusses. It is possible to use *URI shortcuts*, as "Use of web.xml for Tag Libraries" on page 8-14 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 8-20 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 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 8-32 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 8-36.
- 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 8-39.
- 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 8-29.

The remainder of this chapter provides details about these topics.

Overview of Tag Library Changes Between the JSP 1.1 and 1.2 Specifications

The JSP 1.2 specification introduced features for improved tag library support in the following areas:

- Tag library descriptor features

  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 8-5 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 could not have multiple TLD files packaged in a single JAR file. The JSP 1.2 specification supports this, however. See "Tag Handlers" on page 8-20.

- Tag handler features

  Features are summarized in "Summary of Tag Handler Changes Between the JSP 1.1 and 1.2 Specifications" on page 8-5. Tag handler features are described in detail in "Tag Handlers" on page 8-20.

- Tag library validators

This feature was introduced in the JSP 1.2 specification. See "Validation and Tag-Library-Validator Classes" on page 8-36.

- Tag library event listeners

  This feature was also introduced in the JSP 1.2 specification. See "Tag Library Event Listeners" on page 8-39.

- Support for tag attributes of type `Object`

  The JSP 1.2 specification introduced 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 Oracle Application Server 10*g* Release 2 (10.1.2), 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 8-6.

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 prior to Oracle9*i*AS Release 2 (9.0.3). "Tag Library Descriptor Files" on page 8-5 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 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 Parameter Descriptions" on page 3-13 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 in the OC4J demos directory structure. You can find the demos through the following location:

  `http://www.oracle.com/technology/tech/java/oc4j/demos/`

Summary of Tag Handler Changes Between the JSP 1.1 and 1.2 Specifications

The JSP 1.1 specification documented two interfaces to be implemented by tag handlers: `Tag` for tags without bodies, and `BodyTag` for tags with bodies. The JSP 1.2 specification introduced 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 as of the JSP 1.2 specification, 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 introduced 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 prior to Oracle9*i*AS Release 2 (9.0.3). "Tag Handlers" on page 8-20 includes information about these new features.

Tag Library Descriptor Files

A *tag library descriptor* (TLD) file is an XML-style document that defines a tag library and its individual tags. 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 8-11.)

The following sections provide an overview and general information about TLD file syntax and usage, referring ahead to other sections as appropriate for more information about related 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*.

See "Example: Using the IterationTag Interface and a Tag-Extra-Info Class" on page 8-43 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 Parameter Descriptions" on page 3-13 for more information. For pretranslation, use the `ojspc -noTldXmlValidate` option, described in "Option Descriptions for ojspc" on page 7-13.

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 in Oracle Application Server 10*g* Release 2 (10.1.2), 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 pretranslation, if the `ojspc -noTldXmlValidation` flag is not used.

Note: According to the JSP 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 default in Oracle Application Server 10*g* Release 2 (10.1.2) is to 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 8-7.

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 8-10.

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 8-13.
- 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 developers who wish to use the tag library. In particular, a TLD can be processed, such as through an XSLT style sheet, to provide developer documentation from the material in the descriptive elements. This information can be displayed in the help windows of tools such as Oracle 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 8-20 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 8-9.

- Each `<variable>` subelement (if any), with its further subelements, defines a scripting variable. See "Scripting Variables, Declarations, and Tag-Extra-Info Classes" on page 8-32 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 8-33 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 8-34 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 Oracle Application Server 10*g* Release 2 (10.1.2), 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, with traditional syntax:

```
<foosql:query ... >
   select ... where salary > 1000
</foosql:query>
```

Compare this 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 "`&gt;`" 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 5-3 for more information about JSP XML syntax.

Other Key Elements and Their Subelements: validator and listener

The TLD `<validator>` and `<listener>` elements were introduced 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 8-36 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 8-39 for more information.

Tag Library and TLD Setup and Access

The following sections discuss the packaging, placement, and access of tag libraries and their TLD files:

- Overview: Specifying a Tag Library with the taglib Directive
- Specifying a Tag Library by Physical Location
- Packaging and Accessing Multiple Tag Libraries in a JAR File
- Use of web.xml for Tag Libraries
- Oracle Extensions for Tag Library Sharing and Persistent TLD Caching
- Example: Multiple Tag Libraries and TLD Files in a JAR File

Notes:

- It is highly recommended to place TLD files under the application `/WEB-INF` directory. In future releases this will be enforced.
- If you use tag library JAR files at the application level that are intended to supersede JAR files in the well-known tag library location (global level), then you must specify that the application class loader search local classes first. In the application `orion-web.xml` file, in the `<web-app-class-loader>` element, set the `search-local-classes-first` attribute to a value of "`true`".
- In OC4J standalone, if you add a tag library JAR file to the `/WEB-INF/lib` directory while OC4J is running, you must use a `tags_reuse_default` flag value of "`none`" or "`compiletime`" to avoid a `ClassCastException`. You must also force retranslation of relevant JSP pages (such as by touching the `.jsp` files or removing the corresponding `.class` files).

Overview: Specifying a Tag Library with the taglib Directive

This section summarizes the use of `taglib` directives, comparing functionality under the JSP 1.1 specification to functionality under the JSP 1.2 and later specifications.

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.

The JSP 1.1 specification stated that 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 8-12 for more information.

Beginning with 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 8-13 for more information. For information about shortcut URIs, see "Use of web.xml for Tag Libraries" on page 8-14.

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 1-21 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 the JSP 1.1 specification, 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 JSP 1.1 specification mandated that the TLD file must be located and named as follows in the JAR file:

```
META-INF/taglib.tld
```

The JSP 1.1 specification also mandated that the JAR file must be located 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 introduced in 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 8-14.

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.

Adding to these scenarios, the JSP 1.2 specification introduced 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.

A single TLD file in a JAR file can be packaged as `/META-INF/taglib.tld`, or you can use another name as desired. (In JSP 1.1, the `taglib.tld` naming convention was a requirement.)

In a JAR file with multiple TLD files, the TLD files must be uniquely named or be in different subdirectories under `META-INF`.

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.
- To avoid unintended results, each `<uri>` value should 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 an OC4J "well-known" tag library location as described in "Oracle Extensions for Tag Library Sharing and Persistent TLD Caching" on page 8-15. 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 8-14 explains.
- A JSP 1.2-compliant 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 8-18.

Use of web.xml for Tag Libraries

The Sun Microsystems *Java Servlet Specification* describes 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 8-12 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 8-13 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* and the Sun Microsystems *JavaServer Pages Specification* for more information about the `web.xml` deployment descriptor.

Important: Generally speaking, the `<taglib>` element in `web.xml` is required in the case of a TLD file that is located in a JSP "well-known" tag library location and has `<listener>` elements. This is the only way that the TLD file can be found and accessed in order to activate its listeners. This is *not* the case, however, if you use persistent TLD caching. See "Oracle Extensions for Tag Library Sharing and Persistent TLD Caching" on page 8-15 and "Tag Library Event Listeners" on page 8-39.

Oracle Extensions for Tag Library Sharing and Persistent TLD Caching

As an extension of standard JSP "well-known URI" functionality described in the JSP specification, the OC4J JSP container supports the use of one or more directories, known as *well-known tag library locations*, where you can place tag library JAR files to be shared across multiple Web applications.

There is also a persistent caching feature for TLD files, with a global cache for TLD files in any well-known tag library locations, as well as an application-level cache for any application that uses TLD caching.

The use of TLD caching speeds performance at application startup and during JSP page translation. You might typically turn it off, however, under either of the following circumstances:

- Your application does not use tag libraries.

or:

- You have pretranslated the JSP pages and none of the TLD files use `<listener>` elements for tag library event listeners. (See "Tag Library Event Listeners" on page 8-39.)

The following sections provide additional information:

- TLD Caching and Well-Known Tag Library Locations
- TLD Cache Features and Files

TLD Caching and Well-Known Tag Library Locations

TLD caching is enabled or disabled through the `jsp-cache-tlds` attribute of the `<orion-web-app>` element, at a global level through this attribute in the `global-web-application.xml` file, or at an application level through this attribute in the application `orion-web.xml` file.

By default, TLD caching is enabled at a global level through the default setting `jsp-cache-tlds="true"` in `global-web-application.xml`. This is also the default setting in the `orion-web.xml` file of each application, but you can disable TLD caching for any particular application with a setting of `jsp-cache-tlds="false"` in `orion-web.xml`. This overrides the global setting.

Alternatively, you can disable TLD caching at a global level with a "`false`" setting in `global-web-application.xml`, then optionally enable TLD caching for any particular application with a "`true`" setting in `orion-web.xml`.

A setting of "`standard`" searches for TLD files only in `/WEB-INF` or subdirectories other than `/WEB-INF/classes` or `/WEB-INF/lib`. The "`true`" setting, by contrast, searches all application files for TLD files.

Note: By default, `orion-web.xml` inherits its `jsp-cache-tlds` setting from `global-web-application.xml`.

If TLD caching is enabled, you can specify one or more well-known tag library locations using a semicolon-delimited list of directory paths in the `jsp-taglib-locations` attribute of the `<orion-web-app>` element in `global-web-application.xml`. See "OC4J Configuration Parameters for JSP" on page 3-20 for additional information about this attribute.

Important: Use the `jsp-taglib-locations` attribute only in `global-web-application.xml`, not in `orion-web.xml`.

If TLD caching is disabled, the well-known tag library location is limited to a single directory, using functionality that existed prior to the availability of TLD caching. In this case, the well-known location is determined by the `well_known_taglib_loc` JSP configuration parameter. See "JSP Configuration Parameters" on page 3-11 for additional information about this parameter.

In an Oracle Application Server environment, the default well-known location is *`ORACLE_HOME`*`/j2ee/home/jsp/lib/taglib` (assuming *`ORACLE_HOME`* is defined).

Important:

- For any application to pick up files in the well-known location or locations, the directory or directories that are specified in `jsp-taglib-locations` or the directory that is specified in `well_known_taglib_loc` must be added to the `path` attribute setting of the `<library>` element in the OC4J global `application.xml` file in the configuration files directory (`j2ee/home/config` by default in OC4J standalone). See the *Oracle Application Server Containers for J2EE User's Guide* for information about `application.xml`.
- If a TLD file is present both in the well-known location and under the `/WEB-INF` directory of an application, the `/WEB-INF` copy takes precedence and is used.
- If TLD files with the same URI value are present in or under the `/WEB-INF` directory and also in a JAR file in the `/WEB-INF/lib` directory, the decision of which one to use is indeterminate. Avoid this situation.

TLD Cache Features and Files

For any application that uses TLD caching, whether it is enabled at the global level or at the application level, there are two levels of caching, and two aspects of caching at each level.

Caching levels:

- There is a global cache for TLD files that are in JAR files in any well-known tag library locations.
- There is an application-level cache for TLD files under the application `/WEB-INF` directory.

  At the application level, tag library JAR files, which include TLD files, must be in the `/WEB-INF/lib` directory. Individual TLD files can be directly in `/WEB-INF` or in any subdirectory, but preferably not in `/WEB-INF/lib` or `/WEB-INF/classes`.

Caching aspects at each level:

- There is a file containing resource information for the relevant location—the well-known location for the global cache, or `/WEB-INF` or `/WEB-INF/lib` for the application-level cache. Because of this feature, JAR files do not have to be scanned more than once. The file contains two types of entries:
  - There is a list of all resources (tag library JAR files) that includes a timestamp for each resource so that any change to any resource can be detected. There is also an indication ("`true`" or "`false`") of whether each resource includes a TLD file.
  - There is a list of TLD files, where each entry consists of a TLD name, TLD URI value if present, and tag library listeners if present. (See "Tag Library Event Listeners" on page 8-39.)

- There is a serialized DOM representation of each TLD file. Because of this feature, TLD files do not have to be parsed more than once.

The global cache is always located in a directory called `tldcache`, parallel to the configuration directory. The `tldcache` directory contains the following:

- There is a file, `_GlobalTldCache`, that contains resource information, as described above, for any well-known locations.
- There are DOM representations of the TLD files that are in well-known locations. For each TLD file that is in a JAR file in a well-known location, the DOM representation is in a subdirectory according to the name of the JAR file, with a file name according to the name of the TLD file. For example, if `email.tld` is found in `ojsputil.jar` in a well-known location, then its DOM representation would be in the following file (file name `email` in directory `ojsputil_jar`):

  ```
  ORACLE_HOME/j2ee/home/jsp/lib/taglib/persistence/ojsputil_jar/email
  ```

  This is for an Oracle Application Server environment, where *`ORACLE_HOME`* is defined. In OC4J standalone, the `j2ee` directory is relative to where OC4J is installed.

The application-level cache is in the directory indicated by the `jsp-cache-directory` setting in either `global-web-application.xml` or `orion-web.xml`. (See "OC4J Configuration Parameters for JSP" on page 3-20 for information about `jsp-cache-directory`.) This directory contains the following:

- There is a file, `_TldCache`, that contains resource information, as described above, for TLD files under the `/WEB-INF` directory—either in JAR files in `/WEB-INF/lib`, or individually in `/WEB-INF` or any subdirectory, but preferably not `/WEB-INF/lib` or `/WEB-INF/classes`.
- There are DOM representations of the TLD files under `/WEB-INF`. For TLD files that are in JAR files in the `/WEB-INF/lib` directory, the DOM representations go into subdirectories under the directory indicated by `jsp-cache-directory`, in the same type of scheme as described for the global cache. For individual TLD files under `/WEB-INF`, the DOM representations go directly in the `jsp-cache-directory` location.

Notes:

- TLD changes at the global level are reflected only after OC4J is restarted.
- TLD changes at the application level are reflected immediately in an OC4J standalone environment, but only after the application is restarted in an Oracle Application Server environment.
- You can increase the OC4J verbosity level to see information regarding construction of TLD caches and regarding any TLD URIs that are duplicated. Level 4 provides some information; level 5 provides additional information. You can use Oracle Enterprise Manager 10*g* to set the verbosity level. The default level is 3.

Example: Multiple Tag Libraries and TLD Files in a JAR File

This section presents an example of tag library packaging. 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. These 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 the appropriate `taglib` directives, 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

The following sections describe *tag handlers*, which define the semantics of actions that result from the use of custom tags:

- Overview of Tag Handlers
- Attribute Handling, Conversions from String Values
- Custom Tag Processing, with or without Tag Bodies
- Summary of Integer Constants for Body Processing
- Simple Tag Handlers without Iteration
- Simple Tag Handlers with Iteration
- Tag Handlers That Access Body Content
- TryCatchFinally Interface
- Access to Outer Tag Handler Instances

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 8-5.

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 8-43 includes code for a sample tag handler class.

Note: The JSP specification 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 8-30 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 1-17.

- 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

A custom tag, as with a standard JSP tag, might or might not have a body. In the case of a custom tag, even when there is a body, its content might 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 8-7.

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 8-25.

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 8-24.

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 was defined in the `BodyTag` interface. Moving this method definition to the `IterationTag` interface, beginning with the JSP 1.2 specification, 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 8-41.

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 8-24 and "Simple Tag Handlers with Iteration" on page 8-25.

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 8-28.

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 introduced 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 can 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 might 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

To improve performance, 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 Oracle Application Server 10*g* Release 2 (10.1.2), 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 3-11 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 current 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 retranslate the JSP pages.
- The JSP container also supports tag handler reuse in a servlet 2.0 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 uses of the same tag handler within the same page. This method releases state information, with details according to the tag handler implementation. If the tag handler is coded in such a way as to assume a release of state information between tag usages, for example, then a `compiletime_with_release` setting would be appropriate. If you are 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 8-30, the effect is not maximized when `reduce_tag_code` is also `true`.

See "JSP Configuration Parameters" on page 3-11 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
- Scripting Variable Scopes
- Variable Declaration Through TLD variable Elements
- Variable Declaration Through Tag-Extra-Info Classes

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 8-33.) 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 8-33.
- 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 8-34.

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

The JSP 1.1 specification mandated that 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 8-34. The JSP 1.2 specification, however, introduced 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 8-33. 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 8-5 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 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.
- `TagInfo`: An instance of this class contains information about the relevant tag. The class is instantiated from the TLD file and is available only at translation time. `TagInfo` has methods such as `getTagName()`, `getTagClassName()`, `getBodyContent()`, `getDisplayName()`, and `getInfoString()`.

You can refer to the following location for further information:

http://java.sun.com/j2ee/sdk_1.3/techdocs/api/javax/servlet/jsp/tagext/package-summary.html

---

Note: It is uncommon to use `TagInfo` instances in a tag-extra-info implementation, although it might be useful if you want to map a single tag-extra-info class to multiple tag libraries and TLD files, for example.

The following methods of the `TagExtraInfo` class are related:

- `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.

- `void setTagInfo(TagInfo info)`

  Calling this method sets a `TagInfo` instance as an attribute of the tag-extra-info class. This method is typically called by the JSP container.

- `TagInfo getTagInfo()`

  Use this method to retrieve the `TagInfo` attribute of the tag-extra-info class, assuming the `TagInfo` attribute was previously set.

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 10.1.2 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 8-45 for sample code of a tag-extra-info class.

Validation and Tag-Library-Validator Classes

The JSP 1.2 specification introduced 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.

The following sections discuss tag library validation and TLV classes:

- 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. The use of the tag library with which the TLV class is associated is allowed implicitly. In addition, 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 5-11.)
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 5-12 for information.

1. 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 might 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 servlet specification 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. See the *Oracle Application Server Containers for J2EE Servlet Developer's Guide* for additional information about the event listeners.

The JSP 1.2 specification, for convenience in packaging and deploying tag libraries, introduced 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. The following sections describe these features:

- 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 8-11. That section includes information about OC4J well-known tag library locations.

Also, generally speaking, for any TLD 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. The `<taglib>` element in `web.xml` is *not* required, however, if you are using persistent TLD caching (described in "Oracle Extensions for Tag Library Sharing and Persistent TLD Caching" on page 8-15).

End-to-End Custom Tag Examples

The following sections provide complete examples of custom tag usage, including sample JSP pages, tag handler classes, and tag library descriptor files:

- 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://www.oracle.com/technology/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>
JSP 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.

This example includes 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
- TLD file

Note: Sample code here uses extended datatypes in the `oracle.jsp.jml` package. For information, refer to the *Oracle Application Server 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 is to 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 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 JSP specification, use a runtime support mechanism. They are typically portable, not requiring any particular JSP container.

It is also possible, however, 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 might be scenarios where tags using a compile-time mechanism are appropriate, as this section discusses.

General Compile-Time Versus Runtime Considerations

The JSP 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 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 might 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 *Oracle Application Server 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 JSP specification, 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 sections are included:

- Content Type Settings
- JSP Support for Multibyte Parameter Encoding

Note: For detailed information about Oracle Application Server Globalization Support, see the *Oracle Application Server Globalization Guide*.

Content Type Settings

The following sections cover standard ways to statically or dynamically specify the content type for a JSP page. There is also discussion of an Oracle extension method that enables you to specify a non-IANA (Internet Assigned Numbers Authority) character set for the JSP writer object.

- 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 introduced 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"
```

Alternatively, 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 the following location:

`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 Table 9–1.

Table 9–1 Effect of pageEncoding and contentType on Character Encodings

pageEncoding Status	contentType Status	Page Source Encoding Status	Response Encoding Status
Specified	Specified	According to `pageEncoding`	According to `contentType`
Specified	Not specified	According to `pageEncoding`	According to `pageEncoding`
Not specified	Specified	According to `contentType`	According to `contentType`
Not specified	Not specified	According to default	According to 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 9-4.

  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 9-5.

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 3-11 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 9-1.

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 servlet 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

These topics are covered in the following sections:

- Standard setCharacterEncoding() Method
- Overview of Oracle Extensions for Older Servlet Environments

Standard setCharacterEncoding() Method

Beginning 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, which is 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, Oracle provides two alternative mechanisms:

- `oracle.jsp.util.PublicUtil.setReqCharacterEncoding()` static method (preferred)
- `translate_params` configuration parameter (or equivalent code)

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 might 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 sections are included:

- 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 *Oracle Application Server Containers for J2EE Servlet Developer's Guide*.

Background on Servlets

Because JSP pages are translated into Java servlets, a brief review of servlet technology might be helpful. Refer to the Sun Microsystems *Java Servlet Specification* for more information about the concepts discussed here.

For information about the methods this section discusses, refer to Sun Microsystems Javadoc at the following location:

`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 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 extends `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. Multiple separate servlet instances are spawned 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 can expire automatically after a set amount of time or can 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.

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 classloader 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 servlet 2.2 specification, 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 servlet 2.2 specification 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 servlet 2.3 specification introduced 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 *Oracle Application Server 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 `/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 4-1.

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 can 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.

---

1. 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-4.

2. 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 of 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 1-6.)

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-2 for a discussion of the `Servlet` interface and its key methods.

B

Third Party Licenses

This appendix includes the Third Party License for third party products included with Oracle Application Server and discussed in this manual. Topics include:

- Apache HTTP Server

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

Index