Oracle® Containers for J2EE

Support for JavaServer Pages Developer's Guide

10*g* Release 3 (10.1.3)

Part No. B14430-01

January 2006

Oracle Containers for J2EE Support for JavaServer Pages Developer's Guide, 10*g* Release 3 (10.1.3)

Part No. B14430-01

Copyright © 2002, 2006, Oracle. All rights reserved.

Primary Author: Dan Hynes

Contributing Author: Brian Wright

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, Qiang Lin, Song Lin, Jeremy Lizt, Angela 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

The Programs (which include both the software and documentation) contain proprietary information; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the Programs, except to the extent required to obtain interoperability with other independently created software or as specified by law, is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. This document is not warranted to be error-free. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose.

If the Programs are delivered to the United States Government or anyone licensing or using the Programs on behalf of the United States Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the Programs, including documentation and technical data, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement, and, to the extent applicable, the additional rights set forth in FAR 52.227-19, Commercial Computer Software—Restricted Rights (June 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065

The Programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, redundancy and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and we disclaim liability for any damages caused by such use of the Programs.

Oracle, JD Edwards, PeopleSoft, and Retek are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

The Programs may provide links to Web sites and access to content, products, and services from third parties. Oracle is not responsible for the availability of, or any content provided on, third-party Web sites. You bear all risks associated with the use of such content. If you choose to purchase any products or services from a third party, the relationship is directly between you and the third party. Oracle is not responsible for: (a) the quality of third-party products or services; or (b) fulfilling any of the terms of the agreement with the third party, including delivery of products or services and warranty obligations related to purchased products or services. Oracle is not responsible for any loss or damage of any sort that you may incur from dealing with any third party.

Contents

Preface xi

Intended Audience xi
Documentation Accessibility xi
Related Documents xii
Conventions xiii

1 Getting Started with JSP

A Brief Overview of JavaServer Pages Technology 1-1
What is JavaServer Pages Technology? 1-1
Key Advantages of JSP 1-2
How JSP Works 1-2
JSP Translation and Runtime Flow 1-3
Overview of JSP Syntax Elements 1-4
Directives 1-5
page directive 1-5
include directive 1-6
taglib directive 1-6
Scripting Elements 1-6
Declarations 1-7
Expressions 1-7
Scriptlets 1-7
Comments 1-8
JSP Objects and Scopes 1-8
Explicit Objects 1-9
Implicit Objects 1-9
Using an Implicit Object 1-10
Object Scopes 1-10
Standard JSP Action Tags 1-11
jsp:useBean tag 1-12
jsp:setProperty tag 1-13
jsp:getProperty tag 1-14
jsp:param tag 1-14
jsp:include tag 1-14
jsp:forward tag 1-15
jsp:plugin tag 1-16

Bean Property Conversions from String Values 1-16
Typical Property Conversions 1-17
Conversions for Property Types with Property Editors 1-17
Custom Tag Libraries 1-17
Simplified JSP Authoring with the Expression Language 1-18
Overview of the Expression Language Syntax 1-18
JSP Expression Language Syntax 1-18
Expression Language Implicit Objects 1-19
Additional Features of the Expression Language 1-20
Creating and Using Expression Language Functions 1-20
Disabling the Expression Language 1-21
Disabling EL in All JSPs in a Web Application 1-21
Disabling EL in a JSP 1-21
Disabling EL in a Tag File 1-21
JSP Execution Model 1-21
JSP Execution Models 1-21
On-Demand Translation Model 1-22
Pretranslation Model 1-22
JSP Pages and On-Demand Translation 1-22
Requesting a JSP Page 1-23
Directly Requesting a JSP Page 1-23
Indirectly Requesting a JSP Page 1-23

2 The Oracle JSP Implementation

Introduction to OC4J 2-1
What's New in OC4J 2-1
Support for Web Services 2-1
Support for New J2EE 1.4 Application Management and Deployment Specifications 2-2
Support for Oracle Application Server TopLink 2-2
OracleAS Job Scheduler 2-2
New Two-Phase Commit Transaction Coordinator Functionality 2-2
Generic JMS Resource Adapter Enhancements 2-2
Features of OC4J 2-3
J2EE Support 2-3
OC4J Web Communication 2-3
Clustering 2-3
Oracle Value-Added Features for JSP 2-3
Supported Specifications 2-4
Oracle-Specific Features 2-4
Configurable JSP Extensions in OC4J 2-4
Global Includes 2-5
Support for Dynamic Monitoring Service 2-5
JSP Utilities and Tag Libraries Provided with OC4J 2-5
Tags and API for Caching Support 2-6
Support for the JavaServer Pages Standard Tag Library (JSTL) 2-6
JSP Support in Oracle JDeveloper 2-7
Oracle JSP Resource Management Features 2-8

Standard Session Resource Management: HttpSessionBindingListener 2-8
The valueBound() and valueUnbound() Methods 2-8
JDBCQueryBean JavaBean Code 2-8
UseJDBCQueryBean JSP Page 2-10
Advantages of HttpSessionBindingListener 2-11
Overview of Oracle Value-Added Features for Resource Management 2-11

3 Configuring the OC4J JSP Environment

Configuring the OC4J JSP Container 3-1
Summary of JSP Configuration Parameters 3-1
Setting JSP Parameters in Application Server Control Console 3-7
Accessing Application Server Control Console in Standalone OC4J 3-7
Accessing Application Server Control Console in Oracle Application Server 3-7
Setting JSP Parameters in the XML Configuration Files 3-8
Setting Servlet Initialization Paramters 3-8
Setting JSP Configuration Parameters 3-9
Configuring JSP Compilation in OC4J 3-9
Configuring Runtime JSP Retranslation and Reloading in OC4J 3-9
Key JSP-Related Support Files Provided with OC4J 3-10

4 Precompiling JSPs with ojspc

How the ojspc Utility Works 4-1
Overview of Basic ojspc Functionality 4-1
Overview of Batch Pretranslation of WAR Files 4-2
Using ojspc 4-2
Precompiling One or More JSPs 4-3
Precompiling JSPs within a WAR File 4-4
Complete Summary of ojspc Command Line Options 4-4

5 Understanding JSP Translation in OC4J

Features of Generated Code 5-1
General Conventions for Output Names 5-2
Generated Package and Class Names 5-3
Generated Files and Locations 5-4
Oracle JSP Global Includes 5-5
Global Includes File and Examples 5-6
The ojsp-global-include.xml File 5-6
<ojsp-global-include> 5-6
<include ... > 5-6
<into ... > 5-6
Global Include Examples 5-7
Example: Header/Footer 5-7
Example: translate_params Equivalent Code 5-8

6 Working with JSP

Before You Start 6-1
Understanding Application Root Functionality 6-1
Understanding OC4J Classpath Functionality 6-2
Packages Imported By Default in OC4J 6-3
JDK1.4 Issue: Classes Not in Packages Cannot Be Invoked 6-4
General JSP Programming Strategies 6-4
Creating Traditional Versus Scriptless JSP 6-5
Using JavaBeans Versus Scriptlets 6-5
Using Static Includes Versus Dynamic Includes 6-6
Logistics of Static Includes 6-6
Logistics of Dynamic Includes 6-6
Advantages, Disadvantages, and Typical Uses of Dynamic and Static Includes 6-6
Monitoring Your JSP Application 6-7
Managing Heavy Static Content or Tag Library Usage 6-8
Using Method Variable Declarations Versus Member Variable Declarations 6-9
Working with Page Directives 6-10
Page Directives Are Static 6-10
Example 6-10
Example 2 6-10
Duplicate Settings of Page Directive Attributes Are Disallowed 6-10
Workarounds for the 64K Size Limit for Generated Methods 6-12
Following JSP File Naming Conventions 6-12
Understanding JSP Preservation of White Space and Use with Binary Data 6-12
White Space Examples 6-12
Example : No Carriage Returns 6-12
Example 2: Carriage Returns 6-13
Reasons to Avoid Binary Data in JSP Pages 6-13
JSP Best Practices 6-14
Beware of HTTP Sessions 6-14
Avoid Using HTTP Sessions If Not Required 6-14
Always Invalidate Sessions When No Longer In Use 6-14
Pre-translate JSP Pages Using the ojspc Utility 6-14
Ensure Updated Objects Are Re-set on HTTP Sessions 6-15
Un-Buffer JSP Pages 6-15
Forward to JSP Pages Instead of Using Redirects 6-15
Hide JSP Pages from Direct Invocation to Limit Access 6-15
Use JSP-Timeout for Efficient Memory Utilization 6-16
Package JSP Files In EAR File For Deployment 6-16
Working with Servlets 6-16
Invoking a Servlet from a JSP Page 6-17
Passing Data to a Servlet Invoked from a JSP Page 6-17
Invoking a JSP Page from a Servlet 6-17
Passing Data Between a JSP Page and a Servlet 6-18
JSP-Servlet Interaction Samples 6-19
Code for Jsp2Servlet.jsp 6-19
Code for MyServlet.java 6-19

Code for welcome.jsp 6-19
Processing Runtime Errors 6-19
Servlet and JSP Runtime Error Mechanisms 6-20
General Servlet Runtime Error Mechanism 6-20
JSP Error Pages 6-20
JSP Error Page Example 6-20
Code for nullpointer.jsp 6-20
Code for myerror.jsp 6-21

7 Working with Custom Tags

What Are Custom Tags? 7-1
Available Tag Libraries 7-2
When Should You Consider Creating/Using Custom Tag Libraries? 7-2
Eliminating Extensive Java Logic 7-3
Providing Convenient JSP Programming Access to API Features 7-3
Manipulating or Redirecting JSP Output 7-3
Working with Tag Handlers 7-3
What Are Classic Tag Handlers? 7-4
Classic Tag Handler Interfaces 7-4
Custom Tag Processing, with or without Tag Bodies 7-4
Tag Handlers That Access Body Content 7-5
What Are Simple Tag Handlers? 7-7
The SimpleTag Interface 7-7
Using Attributes 7-7
Attribute Handling and Conversions from String Values in Tag Handlers 7-8
Using Scripting Variables in Tags 7-9
Scripting Variable Scopes 7-9
Variable Declaration Through TLD variable Elements 7-10
Variable Declaration Through Tag-Extra-Info Classes 7-11
Access to Outer Tag Handler Instances 7-12
Implementing a Tag Handler 7-13
Creating the Tag Handler Class 7-13
Defining the Tag in the TLD 7-13
Declaring the Tag in a JSP Page 7-15
Using the Tag in a JSP 7-16
OC4J Tag Handler Features 7-16
Disabling or Enabling Tag Handler Reuse (Tag Pooling) 7-16
Enabling or Disabling the Compile-Time Model for Tag Handler Reuse 7-16
When Can the Compile-Time Tag Pooling Model Be Used? 7-17
Code Pattern for the compiletime Tag Pooling Model 7-18
Code Pattern for the compiletime-with-release Tag Pooling Model 7-18
Tag Handler Code Generation 7-18
Working with Tag Files 7-18
What Are Tag Files? 7-19
Tag Body Processing 7-19
Using Attributes in Tag Files 7-20
Exposing Data through Variables in Tag Files 7-20

Using JSP Fragments.......... 7-21
Creating a JSP Fragment.......... 7-21
A Tag File Example.......... 7-22
Implementing a Tag File.......... 7-23
Creating the Tag File.......... 7-23
Packaging Tag Files.......... 7-23
Declaring the Tag File in a JSP.......... 7-24
Sharing Tag Libraries Across Web Applications.......... 7-24
Packaging Multiple Tag Libraries and TLD Files in a JAR File.......... 7-24
Key TLD File Entries.......... 7-25
Key web.xml Deployment Descriptor Entries.......... 7-25
JSP Page taglib Directives for Multiple-Library Example.......... 7-26
Specifying Well-Known Tag Library Locations.......... 7-26
Enabling the TLD Caching Feature.......... 7-27
Understanding the TLD Cache Features and Files.......... 7-28

8 Understanding JSP XML Support in OC4J

Introducing JSP Documents and XML Views.......... 8-1
Working with JSP Documents.......... 8-3
Specifying a Document Root Element.......... 8-4
Declaring Tag Libraries with XML Namespaces.......... 8-4
Using JSP XML Directive Elements.......... 8-5
Example: page Directive.......... 8-5
Example: include Directive.......... 8-5
Using JSP XML Declaration, Expression, and Scriptlet Elements.......... 8-6
Example: JSP Declaration.......... 8-6
Example: JSP Expression.......... 8-6
Example: JSP Scriptlet.......... 8-6
Using JSP XML Standard Action and Custom Action Elements.......... 8-6
Including Template and Dynamic Template Content.......... 8-7
Sample Comparison: Traditional JSP Page Versus JSP XML Document.......... 8-8
Sample Traditional JSP Page.......... 8-8
Sample JSP Document.......... 8-9
Understanding the JSP XML View.......... 8-10
Transformation from a JSP Page to the XML View.......... 8-10
The jsp:id Attribute for Error Reporting During Validation.......... 8-11
Example: Transformation from Traditional JSP Page to XML View.......... 8-11
Traditional JSP Page.......... 8-11
XML View of JSP Page.......... 8-12

9 JSP Globalization Support in Oracle

Content Type Settings.......... 9-1
Content Type Settings in the page Directive.......... 9-1
Dynamic Content Type Settings.......... 9-4
Oracle Extension for the Character Set of the JSP Writer Object.......... 9-4
JSP Support for Multibyte Parameter Encoding.......... 9-5
Standard setCharacterEncoding() Method.......... 9-5

A Third Party Licenses

Apache A-1
The Apache Software License A-1
License A-1
Notice A-5

Index

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 Containers for J2EE (OC4J).

The OC4J Web container in is a complete implementation of the *JavaServer Pages Specification, Version 2.0* and *Servlet Specification, Version 2.4*.

This preface contains the following sections:

- Intended Audience
- Documentation Accessibility
- 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
- How to configure their Web server and servlet environments
- HTML
- Java
- Oracle JDBC (for JSP applications accessing Oracle Database)

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. Accessibility 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 more information, visit the Oracle Accessibility Program Web site at

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

Accessibility of Code Examples in Documentation

Screen readers 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, some screen readers 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.

TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

Related Documents

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

- *Oracle Containers for J2EE Configuration and Administration Guide*

  This book provides guidelines and instructions on using OC4J in a standalone development or production environment.

- *Oracle 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 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 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 Containers for J2EE Security Guide*

This document (not to be confused with the *Oracle Application Server 0g 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.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Getting Started with JSP

This chapter introduces and reviews standard features and functionality of JavaServer Pages (JSP) technology, then concludes with a discussion of JSP execution models. For further general information, consult the *JavaServer Pages Specification, Version 2.0* published by Sun Microsystems.

The chapter contains the following sections:

- A Brief Overview of JavaServer Pages Technology
- Overview of JSP Syntax Elements
- Simplified JSP Authoring with the Expression Language
- JSP Execution Model

> **Note:** The Sample Applications chapter available in previous releases has been removed.
>
> Sample code and applications are available from the following location on the Oracle Technology Network:
>
> `http://www.oracle.com/technology/sample_code/index.html`

A Brief Overview of JavaServer Pages Technology

A quick overview of JSP is provided in the following sections:

- What is JavaServer Pages Technology?
- JSP Translation and Runtime Flow
- Key Advantages of JSP

What is JavaServer Pages Technology?

In simple terms, JavaServer Pages (JSP) technology makes it possible for dynamically-generated content to be displayed in a Web browser. JSP pages comprise the presentation layer for any Web-based application running in the Oracle Application Server environment, providing the interface into the application's business logic and processing power.

A JSP page is simply a text file containing two types of text markup:

- HTML or XML, used to format static content such as page layout and template text; and

- JSP syntax elements and possibly embedded Java code, which provide the dynamic content.

Ease of development allows rapid implementation. With the latest release, it is not even necessary for JSP page authors to have a strong understanding of Java.

JavaServer Pages require a Web container that supports JSP page translation and execution. This Web container is provided as part of the Oracle Containers for J2EE (OC4J). See Chapter 2, "The Oracle JSP Implementation" for more on OC4J functionality.

JSP is a key technology of the Java 2 Platform, Enterprise Edition (J2EE) architecture specified by Sun Microsystems. The OC4J Web container is fully compliant with Sun's JSP 2.0 and Servlet 2.4 specifications.

Key Advantages of JSP

For most situations, there are at least three general advantages to using JSP pages instead of servlets:

- **Ease of coding**

  JSP syntax provides a shortcut for coding dynamic Web pages, typically requiring much less code than equivalent servlet code. The JSP translator also automatically handles some servlet coding overhead for you, such as implementing standard JSP or servlet interfaces and creating HTTP sessions.

- **Separation of static content and dynamic content**

  JSP technology attempts to allow some separation between the HTML code development for static content, and the Java code development for business logic and dynamic content. This makes JSP programming accessible and attractive to Web designers, as it simplifies the division of maintenance responsibilities between presentation and layout specialists and Java developers.

- **Reuse of business logic components**

  JSP technology is designed to facilitate the use of reusable components such as JavaBeans and Enterprise JavaBeans (EJBs). JSP tag libraries, typically supplied with J2EE applications, provide additional coding convenience.

How JSP Works

The dynamic nature of a JSP page is enabled through JSP elements embedded within the HTML (or other markup code, such as XML) of your Web pages. These elements provide access to external Java components, such as JavaBeans and Enterprise JavaBeans (EJB), that provide a Web application's business logic and processing power. These components can in turn directly or indirectly access a database or other EIS.

A JSP page is translated into a Java servlet, typically at the time that it is requested from a client. The JSP translator is triggered by the `.jsp` file name extension in a URL. The translated page is then executed, processing HTTP requests and generating responses similarly to any other servlet. Note that coding a JSP page is dramatically more convenient than coding the equivalent servlet.

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

Here is the code for a simple JSP page, `welcomeuser.jsp`:

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

```
<HTML>
<HEAD><TITLE>The Welcome User JSP</TITLE></HEAD>
<BODY>
<H3>Welcome ${param.user}!</H3>
<P><B> Today is ${Date}. Have a fabulous day! :-)</B></P>
<B>Enter name:</B>
<FORM METHOD=GET>
<INPUT TYPE="text" NAME="user" SIZE=5>
<INPUT TYPE="submit" VALUE="Submit name">
</FORM>
</BODY>
</HTML>
```

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

```
Welcome Amy!

Today is Wed Jun 2 3:42:23 PDT 2000. Have a fabulous day! :-)
```

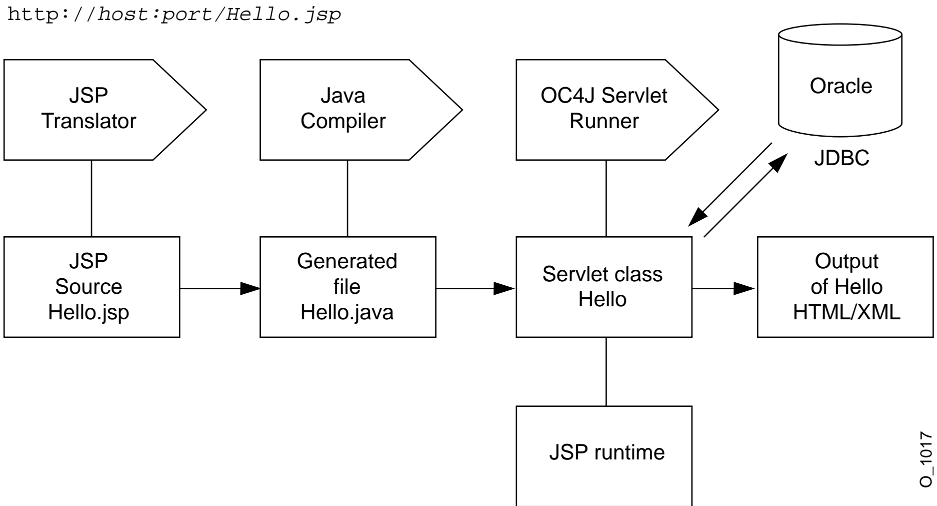
JSP Translation and Runtime Flow

Figure 1–1 illustrates a conceptual overview of the flow of execution when a user calls a JSP page by specifying its URL in the browser. Assume that `Hello.jsp` accesses a database.

Because of the `.jsp` file name extension, the following steps occur automatically:

1. The JSP translator is invoked, translating `Hello.jsp` and producing the file `Hello.java`.
2. The Java compiler is invoked, creating `Hello.class`.
3. `Hello.class` is executed as a servlet, using the JSP runtime library.
4. The `Hello` class accesses the database through JDBC, as appropriate, and sends its output to the browser.

Figure 1–1 JSP Translation and Runtime Flow



Overview of JSP Syntax Elements

The example below illustrates how HTML markup and JSP elements are used together to provide static and dynamic content in a typical JSP. The dynamic content is written in JSP 2.0 syntax, which is fully supported by the OC4J JSP container.

> **Note:** The JSP 2.0 specification supports an XML-compatible JSP syntax as an alternative to the traditional syntax. This allows you to produce JSP pages that are syntactically valid XML documents. The XML-compatible syntax is described in Chapter 8, "Understanding JSP XML Support in OC4J".

The JSP displays all of the phone numbers stored in the database for the employee specified in the HTTP request. The code creates a JavaBean object containing the employee's phone numbers as a Map of key/value pairs. The JSP iterates over the phone numbers, displaying each key and its value in an HTML table.

```
<%@ page contentType="text/html; charset=UTF-8"; import="mypkg.*" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
    prefix="c" %>
<html>
 <head><title>Phone List</title></head>
 <body>

 <jsp:useBean id="employee" scope="application" class="mypkg.Employee"/>
  <jsp:setProperty name="employee" property="empUserId" param="employeeId"/>
 <c:set var="empName" value="${employee.fullName}" />
  <h2>Current Phone Numbers for ${empName}</h2>
   <c:if test ="${!empty employee.phoneNumbers}>
    <table>
      <tr>
       <th>Phone Type:</th><th>Number:</th>
      </tr>
     <c:forEach var="entry" items="{$employee.phoneNumbers}">
      <tr>
       <td>${entry.key}</td>
       <td>${entry.value}></td>
      </tr>
     </c:forEach>
    </table>
   </c:if>
   <c:if test="${empty employee.phoneNumbers}">
    <c:out value="No phone numbers were found for ${empName}">
   </c:if>
  </body>
</html>
```

The JSP elements used in the example are as follows:

- *Directives* provide the OC4J container with instructions specifying how the JSP is to be processed.

  The `page` directive (`<%@ page ... %>`) used in this example specifies the content type returned by the page and imports the `mypkg` package for use. The `taglib` directive (`<%@ taglib ... %>`) imports a custom tag library for use in the page; in this case, the JavaServer Pages Standard Tag Library (JSTL) core library.

- *Standard action tags* provide easy mechanisms for invoking certain common tasks, such as forwarding execution to another JSP or creating an object and accessing its properties.

  In this example, the `jsp:useBean` standard action returns an `Employee` JavaBean instance.

  The `jsp:setProperty` tag sets the value of a bean property; in this case, it sets the `empUserId` property to the value of the `employeeId` request parameter.

- Custom tags are similar to standard action tags, except that they are created and packaged by software vendors and page authors - such as yourself. Custom tags allow JSP pages to access logic provided by reusable Java classes, removing the need to embed Java code directly in pages themselves.

  Note how `c:forEach` tags are used to iterate over the user's phone numbers, while `c:if` tags are used to test whether phone numbers are found for the given user, and to print a message if not.

  Every custom tag belongs to a *tag library,* indicated by the tag prefix. Here, the prefix `c:` identifies the tags used as belonging to the JSTL `core` tag library. See Chapter 7, "Working with Custom Tags" for details on creating and using custom tags.

- JSP expression language (EL) expressions, identifiable by the `${ ... }` syntax, provide easy access to objects such as JavaBeans and their properties.

  The EL is an alternative to the Java expressions used in traditional JSP syntax. For example, the EL expression`${employee.fullName}` is equivalent to the JSP scripting expression `<% = employee.getFullName() %>`.

  See "Simplified JSP Authoring with the Expression Language" on page 1-18 for details.

The following section discusses the basic syntax of JSP, including directives, scripting elements, and standard action tags, and provide a few examples. There is also discussion of bean property conversions. For additional information on JSP 2.0 syntax, see the Sun Microsystems *JavaServer Pages Specification, version 2.0.*

> **Note:** This section describes standard JSP syntax. For information about JSP XML syntax and JSP XML documents, see Chapter 8, "Understanding JSP XML Support in OC4J".

Directives

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

```
<%@ directive attribute="value" 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 un-buffers the page:

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

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 1-23 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.
- The `include` directive can be used only between files in the same servlet context (application).

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 1-17, and discussed in detail in Chapter 7, "Working with Custom Tags".

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 f=0.0; %>
```

This example declares a member variable, `f`. In the servlet class code generated by the JSP translator, `f` 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 "Using Static Includes Versus Dynamic Includes" on page 6-6 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.

The following example assumes the use of a JavaBean instance, `pageBean`:

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

Note how the 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 conditionally executed within the `if` and `else` branches (inside the Java brackets set out in the scriptlets).

This next example adds more Java code to the scriptlets.:

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

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 "Using Method Variable Declarations Versus Member Variable Declarations" on page 6-9.

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 JSP Action Tags" on page 1-11, 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 1-10.

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 Web 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.

- `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, which extends `javax.servlet.jsp.JspContext` as of JSP 2.0.

  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.

- `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, to which processing is forwarded when an exception is thrown from another JSP page. These error pages must have the `page` directive `isErrorPage` attribute set to `true`.

  The implicit `exception` object is a `java.lang.Throwable` 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 "Processing Runtime Errors" on page 6-19.

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 Web 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 JSP Action Tags" below.)

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

Standard JSP Action 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:tagattr="value" attr2="value2" ... attrN="valueN">
...body...
</jsp:tag>
```

Alternatively, if there is no body:

```
<jsp:tag attr="value", ..., attrN="valueN" />
```

The most commonly-used JSP standard action tags are introduced and briefly discussed below:

- `jsp:usebean`
- `jsp:setProperty`
- `jsp:getProperty`
- `jsp:param`
- `jsp:include`
- `jsp:forward`
- `jsp:plugin`

Note: The following tags are covered elsewhere in this book:

- The `doBody` tag is discussed in "Tag Body Processing" on page 7-19.
- The attribute and invoke tags are covered in "Using JSP Fragments" on page 7-21.
- The `body`, `element` and `text` tags are covered in "Including Template and Dynamic Template Content" on page 8-7.

---

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 1-17. Scopes are discussed in "JSP Objects and Scopes" on page 1-8.

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 Web 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.

Use the `class` attribute to specify a class that can be instantiated, if necessary, by the Web container. The class cannot be abstract and must have a no-argument constructor.

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 Web container creates the instance by using the `instantiate()` method of the `java.beans.Beans` class.

Use the `type` attribute to specify a type that cannot be instantiated by the Web 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 Web 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.

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 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 1-16 for additional information.

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 in "Configuring the OC4J JSP Container" on page 3-1. (Continuing was the default behavior in previous releases. As of the OC4J 9.0.4 implementation, however, the default behavior is to stop on errors.)
- 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:params` 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 Web container support either a "`true`" or "`false`" setting, with "`false`" being the default.

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 "Using Static Includes Versus Dynamic Includes" on page 6-6 for a comparison between the two.
- 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. You can 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 included when using `jsp:param` in a `jsp:include` or `jsp:forward` action.)

Note the use of the `jsp:fallback` tag to delimit alternative text to execute if the plugin cannot run.

The following example shows the use of an applet plugin:

```
<jsp:plugin type=applet code="Sample.class" codebase="/html" >
   <jsp:params>
      <jsp:param name="sample" value="samples/sample01" />
   </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 Web container implementations.

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

For information about these topics, see Chapter 7, "Working with Custom Tags".

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

Simplified JSP Authoring with the Expression Language

The *JSP expression language (EL)* greatly simplifies JSP authoring by removing the need to use embedded Java scriptlets and expressions to access request parameters or application data stored in JavaBeans.

The EL was originally introduced as part of the JavaServer Pages Standard Tag Library (JSTL) version 1.0. With the JSP 2.0 release, the EL was made an integral part of the JSP specification, dramatically improving its data-access capabilities.

The JSP 2.0-compliant OC4J container understands EL expressions implemented in the following manner:

- As values for attributes in any standard action (such as `jsp:useBean`) or custom tag that accept runtime expressions; and
- In static template text, such as HTML or non-JSP elements. In this usage, the value of the expression within the text is evaluated and inserted into the current output. However, it must be noted that an expression will not be evaluated if the body of the tag is declared to be `tagdependent`.

As an example, consider the following use of the JSTL `c:if` tag to pick out steel-making companies from a company list:

```
<c:if test="${company.industry == 'steel'}">
   ...
</c:if>
```

Overview of the Expression Language Syntax

This section summarizes the expression language syntax and documents how to enable EL evaluation in your OC4J JSP applications.

Note that although the EL has its own syntax, it is not a general purpose programming language; rather, it is a data access mechanism intended to simplify the lives of JSP authors.

JSP Expression Language Syntax

The expression language has its own syntax, partially based on JavaScript syntax. The following list offers a brief summary of key syntax features of the JSP expression language. This is followed by a few simple examples.

- Invocation

  The expression language is invoked through `${expression}` syntax. The most basic semantic is that invocation of a named variable `${foo}` yields the same result as the method call `PageContext.findAttribute(foo)`.

- Data structure access

  To access named properties within JavaBeans and within collections such as lists, maps, and arrays, the expression language supports the "`.`" and "`[]`" operators.

The "." construct allows access to properties whose names are standard Java identifiers. For example, `employee.phones.cell` is equivalent to `employee.getPhones().getCell()` in Java syntax.

The "[]" construct is for more generalized access, such as for accessing arrays or lists. However, for valid Java identifiers it is equivalent to the "." construct. For example, the expressions `employee.phoneNumbers` and `employee["phoneNumbers"]` yield the same result.

- Relational operators

  The expression language supports the relational operators == (or `eq`), != (or `ne`), < (or `lt`), > (or `gt`), <= (or `le`), >= (or `ge`).

- Arithmetic operators

  The expression language supports the arithmetic operators +, -, *, / (or `div`), % (or `mod`, for remainder or modulo).

- Logical operators

  The expression language supports the logical operators && (or `and`), || (or `or`), ! (or `not`), `empty`.

Basic Example

The following example shows a basic invocation of the expression language, including the relational "<=" (less than or equal to) operator.

```
<c:if test="${auto.price <= customer.priceLimit}">
   The <c:out value="${auto.makemodel}"/> is in your price range.
</c:if>
```

Accessing Collections Example

The following example shows use of the "." and "[]" constructs. Here, `catalogue` is a `Map` object containing the description of products, while `preferences` is a `Map` object containing a particular user's preferences.

```
Item:
<c:out value="${catalogue[productId]}"/>
   Delivery preference:
<c:out value="${user.preferences['delivery']}"/>
```

Expression Language Implicit Objects

The expression language provides the following implicit objects:

- `pageScope`: Allows access to page-scope variables.
- `requestScope`: Allows access to request-scope variables.
- `sessionScope`: Allows access to session-scope variables.
- `applicationScope`: Allows access to application-scope variables.
- `pageContext`: Allows access to all properties of the page context of a JSP page.
- `param`: A Java `Map` object containing request parameters typically accessed using the `request.getParameter()` method. The expression `${param["foo"]}` or the equivalent `${param.foo}` both return the first string value associated with the request parameter `foo`.
- `paramValues`: Use `paramValues["foo"]`, for example, to return an array of all string values associated with request parameter `foo`.

- `header`: As with `param`, you can use this object to access the first string value associated with a request header.
- `headerValues`: Similarly to using `paramValues`, you can use this to access all string values associated with a request header.
- `initParam`: Allows access to context initialization parameters.
- `cookie`: Allows access to cookies received in the request.

Additional Features of the Expression Language

The expression language also offers the following features:

- It can provide default values where failure to evaluate an expression is considered to be recoverable.
- Where application data might not exactly match the type expected by a tag attribute or expression language operator, there are rules to convert the type of the resulting value to the expected type.

Creating and Using Expression Language Functions

The expression language allows you to define static methods known as *functions* that can be invoked within EL expressions.

Creating or using a function is similar to creating or using a custom tag. In fact, the JSTL contains six custom tags that are actually expression language functions. See Chapter 7, "Working with Custom Tags" for details on custom tag implementation.

A function must be implemented as a public static method within a public Java class. The following example paraphrases the static method for the JSTL `fn:length` function available in the Jakarta Taglibs Standard library:

```
public static int length(Object obj)
throws JspTagException {
...
}
```

Classes containing function methods are grouped into *tag libraries*, similar to custom tags. Each function's signature and the mapping to the public class containing its corresponding method are added to the library's *tag library descriptor (TLD)* file. For example:

```
<function>
 <description>
  Returns the number of items in a collection or the number of
  characters in a string.
 </description>
 <name>length</name>
 <function-class>
  org.apache.taglibs.standard.functions.Functions
 </function-class>
 <function-signature>
  int length(java.lang.Object)
 </function-signature>
</function>
```

To use an EL function, a JSP must import the appropriate tag library using a`taglib` directive. Here the page imports the JSTL `functions` library which contains the Java class implementing the `fn:length` function:

```
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
```

Finally, the function can be invoked within an EL expression in the JSP. Here the scoped variable `employees` is a collection of `Employee` objects.

```
There are ${fn:length(employees)} employees listed in the
database.
```

Disabling the Expression Language

It is possible to disable or deactivate the expression language to allow a pattern written in EL syntax to be passed through a JSP, without being evaluated as an EL expression. The EL can be disabled at either the Web application or individual JSP level. Tag files can also be instructed to ignore EL expressions.

Note that when EL is disabled, the pattern `\$` will not be recognized as a quote, whereas it will be recognized as such if EL is enabled.

Disabling EL in All JSPs in a Web Application

To disable EL for all JSPs in an application, add the following <jsp-property-group> to the application's `web.xml` Web application descriptor file.

```
<jsp-property-group>
  <url-patter>*.jsp</url-pattern>
  <el-ignored>true</el-ignored>
</jsp-property-group>
```

Disabling EL in a JSP

To disable EL evaluation in a JSP page, set the `isELIgnored` attribute of the `page` directive to `true` in the JSP.

Disabling EL in a Tag File

To disable EL evaluation in a tag file, set the `isELIgnored` attribute of the `tag` directive to `true` in the tag file.

JSP Execution Model

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) and error processing.

JSP Execution Models

There are two distinct execution models for JSP pages:

- In most implementations and situations, the Web 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 Web 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 Web 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`.

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.

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 Chapter 4, "Precompiling JSPs with ojspc".

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 Web container.
3. The Web 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 Web container) that implements the `javax.servlet.jsp.HttpJspPage` interface. The servlet class is referred to as the ***page implementation class***. This document will refer to instances of page implementation classes as ***JSP page instances***.

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

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

```
http://host:port/myapproot/dir/HelloWorld.jsp
```

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

Note: General servlet and JSP invocation are discussed in the *Oracle 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/dir/HelloWorld.jsp" /a>
```

The application-relative path in a JSP statement is:

```
<jsp:include page="/dir/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 JSP Action Tags" on page 1-11 discusses the `jsp:include` and `jsp:forward` statements.)

2

The Oracle JSP Implementation

This chapter provides details on the functionality implemented by the Web container provided with Oracle Containers for J2EE (OC4J), a component of the Oracle Application Server. 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 Containers for J2EE JSP Tag Libraries and Utilities Reference*).

The following sections are included:

- Introduction to OC4J
- JSP Support in Oracle JDeveloper
- Oracle Value-Added Features for JSP

Introduction to OC4J

Oracle Containers for J2EE 10*g* Release 3 (10.1.3), or OC4J, provides a complete Java 2 Enterprise Edition (J2EE) 1.4-compliant environment. OC4J Standalone is for use in development environments and in small-to-medium scale production environments.

OC4J is written entirely in Java and executes on the Java Virtual Machine (JVM) of the standard Java Development Kit (JDK). You can run OC4J on the standard JDK that exists on your operating system.

The OC4J documentation assumes that you have a basic understanding of Java programming, J2EE technology, and Web and EJB application technology. This includes deployment conventions such as the `/WEB-INF` and `/META-INF` directories.

What's New in OC4J

Release 3 (10.1.3) of Oracle Containers for J2EE includes a number of new features and enhancements, described below.

Support for Web Services

OC4J provides full support for Web Services in accordance with the J2EE 1.4 standard, including JAX-RPC 1.1. Web Services interoperability is also supported.

- EJB 2.1 Web services end point model
- JSR 109 client and server deployment model
- CORBA Web services: Support for wrapping existing basic CORBA Servants as Web services and auto-generating WSDL from IDL

- Support for source code annotations to customize Web services behavior such as invocation and ending styles (RPC/literal, RPC/encoded, Doc/literal); customizing the Java to XML mapping; enforcing security.
- Database and JMS Web services

Support for New J2EE 1.4 Application Management and Deployment Specifications

OC4J supports the following specifications defining new standards for deploying and managing applications in a J2EE environment.

- The *Java Management Extensions (JMX) 1.2* specification, which allows standard interfaces to be created for managing resources, such as services and applications, in a J2EE environment. The OC4J implementation of JMX provides a JMX client that can be used to completely manage an OC4J server and applications running within it.
- The *J2EE Management Specification (JSR-77)*, a sub-set of JMX that allows standard interfaces to be created for managing applications in a J2EE environment.
- The *J2EE Application Deployment API (JSR-88)*, which defines a standard API for configuring and deploying J2EE applications and modules into a J2EE-compatible environment. The OC4J implementation includes the ability to create and/or edit a deployment plan containing the OC4J-specific configuration data needed to deploy a component into OC4J.

Support for Oracle Application Server TopLink

Oracle Application Server TopLink is an advanced, object persistence framework for use with a wide range of Java 2 Enterprise Edition (J2EE) and Java application architectures. OracleAS TopLink includes support for the OC4J Container Managed Persistence (CMP) container and base classes that simplify Bean Managed Persistence (BMP) development.

OracleAS Job Scheduler

The OracleAS Job Scheduler provides asynchronous scheduling services for J2EE applications. Its key features include capabilities for submitting, controlling and monitoring *jobs*, defined as a unit of work that executes when the work is performed.

New Two-Phase Commit Transaction Coordinator Functionality

The new Distributed Transaction Manager in OC4J can coordinate two-phase transactions between any type of XA resource, including databases from Oracle as well as other vendors and JMS providers such as IBM WebsphereMQ. Automatic transaction recovery in the event of a failure is also supported.

Generic JMS Resource Adapter Enhancements

The Generic JMS Resource Adapter can now be used as an OC4J plug-in for OracleAS JMS that ships with the current version of OC4J as well as for IBM Websphere MQ JMS version 5.3.

Support for lazy transaction enlistment has been added so that JMS connections can be cached and still be able to correctly participate in global transactions.

Finally, the Generic JMS Resource Adapter now has better error handling. Endpoints now automatically retry after provider or system failures, and `onMessage()` errors are handled correctly.

Features of OC4J

The following features are provided with the latest release of OC4J.

J2EE Support

OC4J supports and is certified for the standard J2EE APIs, as listed in Table 2–1

Table 2–1 J2EE APIs Supported by OC4J

J2EE Standard APIs	Version Supported By OC4J
JavaServer Pages (JSP)	2.0
Servlets	2.4
Enterprise JavaBeans (EJB)	2.1
Java Transaction API (JTA)	1.0
Java Message Service (JMS)	1.1
Java Naming and Directory Interface (JNDI)	1.2
Java Mail	1.1.2
Java Database Connectivity (JDBC)	2.0 Extension
Oracle Application Server Java Authentication and Authorization Service (JAAS) Provider	1.0
J2EE Connector Architecture	1.5
Java API for XML-Based RPC (JAX-RPC)	1.1
SOAP with Attachments API for Java (SAAJ)	1.2
Java API for XML Registries (JAXR)	1.0.5

OC4J Web Communication

OC4J supports HTTP and HTTPS communications natively without the use of the Oracle HTTP Server.

The default Web site is defined in the `http-web-site.xml` file, which specifies the default HTTP listener on port `8888`. Additional Web sites may be defined on different ports using variations of this file. See the *Oracle Containers for J2EE Configuration and Administration Guide* for instructions on creating additional Web sites in OC4J.

Clustering

OC4J provides support for HTTP session and stateful session Enterprise JavaBean replication and load balancing across a cluster of OC4J instances. However, cluster configuration and management is completely manual in this release, through editing of the OC4J-specific application configuration file. See the *Oracle Containers for J2EE Configuration and Administration Guide* for details.

Oracle Value-Added Features for JSP

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

- Features that are Oracle-specific
- Features implemented through custom tag libraries, custom JavaBeans, or custom classes that are generally portable to other JSP environments
- 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 Containers for J2EE JSP Tag Libraries and Utilities Reference* .

Supported Specifications

The JSP container provided with OC4J is in full compliance with the following specifications published by Sun Microsystems:

- *JavaServer Pages Specification, Version 2.0*
- *Java Servlet Specification, Version 2.4*
- *JSR-045: Debugging Support for Other Languages*

Feel free to consult these specifications for more details.

Oracle-Specific Features

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

Configurable JSP Extensions in OC4J

In addition to JSP 2.0 compliance, the OC4J Web container includes the following notable features.

Also see "Oracle Value-Added Features for JSP" on page 2-3.

- 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 disabled by default.

- Mode flag for extra imports

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

- "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.

- 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.

The following features are also supported:

- 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.

- 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 Tag Handler Reuse (Tag Pooling)" on page 7-16.

- Output mode for null output

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

Global Includes

The OC4J Web 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 Web 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 5-5.

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 Web 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

Standard configuration for these servlets is in the OC4J `application.xml` and `default-web-site.xml` configuration file. Use the Oracle Enterprise Manager 10*g* Application Server Control Console to access DMS, display DMS information, and, if appropriate, alter DMS configuration.

See the *Oracle Application Server Performance Guide* for precise definitions of the JSP metrics and detailed instructions for viewing and analyzing them.

JSP Utilities and Tag Libraries 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 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
- 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

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 Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Support for the JavaServer Pages Standard Tag Library (JSTL)

The OC4J JSP implementation supports the JavaServer Pages Standard Tag Library (JSTL), as specified in the Sun Microsystems *JavaServer Pages Standard Tag Library* specification.

> **Note:** The JSTL distribution is no longer installed in the `ORACLE_HOME/j2ee/home/jsp/lib/taglib` directory within OC4J.
>
> See "Sharing Tag Libraries Across Web Applications" on page 7-24 for instructions on sharing tag libraries across deployed Web applications.

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:

- 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 I8N-capable internationalization and formatting

  (The term "I8N" 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, refer to the *Oracle 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) and other tag libraries provided with OC4J pre-date JSTL and have areas of duplicate functionality.

For standards compliance, it is generally advisable to use JSTL instead of the custom libraries.

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.

JSP Support in Oracle JDeveloper

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 Web 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

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

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

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

(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 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.

Oracle JSP Resource Management Features

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.

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).

"Oracle Value-Added Features for JSP" 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 2-10), 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() +
            " 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.

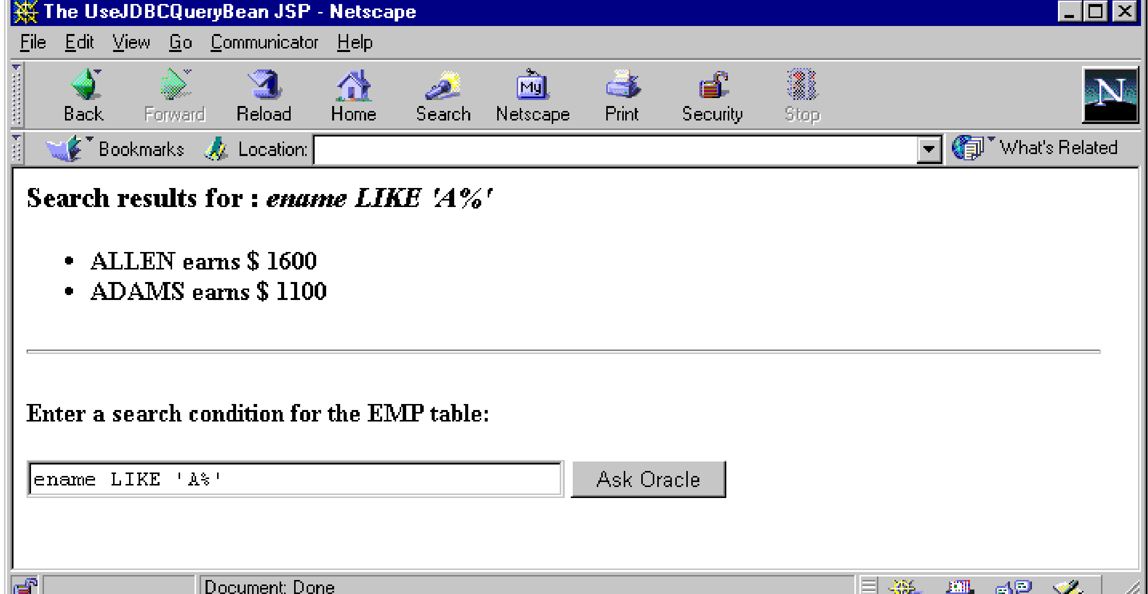
UseJDBCQueryBean JSP Page

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

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

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

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

<% String searchCondition = request.getParameter("searchCond");
   if (searchCondition != null) { %>
      <H3> Search results for : <I> <%= searchCondition %> </I> </H3>
      <%= queryBean.getResult() %>
      <HR><BR>
<% }  %>

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

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

</BODY>
</HTML>
```

Following is sample input and output for this page:



Advantages of HttpSessionBindingListener

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

Overview of Oracle Value-Added Features for Resource Management

OC4J JSP provides the `JspScopeListener` interface for managing application-scope, session-scope, request-scope, or page-scope resources in OC4J.

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

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

3

Configuring the OC4J JSP Environment

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

The following sections are included in this chapter:

- Configuring the OC4J JSP Container
- Configuring JSP Compilation in OC4J
- Configuring Runtime JSP Retranslation and Reloading in OC4J
- Key JSP-Related Support Files Provided with OC4J

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, as any Java code in a JSP page is executed within the Web server.

Configuring the OC4J JSP Container

This section explains your options for configuring the OC4J JSP container, with a focus on configuring the container for JSP development. It includes the following topics:

- Summary of JSP Configuration Parameters
- Setting JSP Parameters in Application Server Control Console
- Setting JSP Parameters in the XML Configuration Files

Summary of JSP Configuration Parameters

A number of parameters for configuring the JSP container environment are available in OC4J. Table 3–1 summarizes the supported configuration parameters.

Parameters that are listed in the **Application Server Control Console JSP Property** column of this table can be set through the JSP Container Properties page of the Oracle Enterprise Manager 10*g* Application Server Control Console management interface provided with OC4J. See "Setting JSP Parameters in Application Server Control Console" on page 3-7 for details.

Parameters that are not exposed through Application Server Control Console can be set directly in the `global-web-application.xml` configuration file, which is located in the `ORACLE_HOME/j2ee/home/config` directory by default. The JSP

parameters persisted in this file are the default values inherited by all Web modules running within the OC4J server instance.

You can override any of the global settings for a specific Web application by modifying the Web module's `web-xml` or `orion-web.xml` deployment descriptors. Note that these files must be edited manually; Application Server Control Console does not provide any editing capabilities.

See "Setting JSP Parameters in the XML Configuration Files" on page 3-8 for details on setting parameters in the `global-web-application.xml` or `orion-web.xml` files.

Table 3–1 JSP Environment Configuration Parameters

Application Server Control Console JSP Property	XML Parameter	Description
(None)	`check_page_scope` `<init-param>`	Set to `true` to enable page-scope checking by `JspScopeListener` within OC4J. The default is `false`. See the *Oracle Containers for J2EE JSP Tag Libraries and Utilities Reference* for detailed information about the `JspScopeListener` utility.
(None)	`debug` `<init-param>`	Set to enable JSR-45 debugging support. Valid values are: ■ `file`: Generates an SMAP file. ■ `class`: Embeds the debugging in the generated class file. ■ `none` (default): No debugging information is generated
Debug Mode	`debug_mode` `<init-param>`	Set to `true` to print the stack trace when certain runtime exceptions occur. The default is `false`. When this parameter 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.
External Resource for Static Content	`external_resource` `<init-param>`	Set to `true` to instruct the JSP translator to place static content within a JSP page into a Java resource file instead of into the generated page implementation class during translation. The default is `false`. The translator places the resource file into the same directory as generated class files. The resource file name is based on the JSP page name, with the `.res` suffix. The translation of `MyPage.jsp`, for example, would create `_MyPage.res` in addition to normal output. (The exact implementation might change in future releases.) 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 "Managing Heavy Static Content or Tag Library Usage" on page 6-8.

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

Application Server Control Console JSP Property	XML Parameter	Description
Extra Import Package List	extra_imports <init-param>	Use to import additional packages beyond the JSP defaults described in "Packages Imported By Default in OC4J" on page 6-3. Note that in an XML file, the names can be either comma-delimited or space-delimited. Either of the following is valid: <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>
Accept Duplicate Directive Attributes	forgive_dup_dir_ attr <init-param>	Set to `true` to avoid JSP translation errors if you have duplicate settings with different values for the same directive attribute within a single JSP translation unit. The default is `false`.
Validate XML	xml_validate <init-param>	Set this boolean to `true` to perform XML validation of the `web.xml` file. The default is `false`, meaning that validation of `web.xml` is *not* performed.
When a JSP Changes	main_mode <init-param>	Use to specify whether JSP-generated classes are automatically reloaded or JSP pages are automatically retranslated when changes are made. If enabled, this feature allows new or modified JSPs to be loaded into the OC4J runtime, without requiring the Web application to be redeployed or restarted. See "Configuring Runtime JSP Retranslation and Reloading in OC4J" on page 3-9 for additional information. Valid settings are: ■ `recompile` (default): The container will check the timestamp of the JSP page, retranslate it and reload it if has been modified since it was last loaded. The functionality described for `reload` will also be executed. ■ `reload`: The container 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 JSP pages, from a development environment to a production environment. ■ `justrun`: The container will not perform any timestamp checking, so there is no retranslation of JSP pages or reloading of JSP-generated Java classes. This is the most efficient mode for a production environment, where JSP pages are not expected to change frequently.
(None)	no_tld_xml_ validate <init-param>	Set to `true` to *not* perform XML validation of TLD files. The default is `true`, meaning validation of the XML structure of TLD files is not performed.
(None)	old_include_ from_top <init-param>	Set 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. The default is `false`.

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

Application Server Control Console JSP Property	XML Parameter	Description
Precompile Check	precompile_check <init-param>	Set to true to check the HTTP request for a standard jsp_precompile setting. The default is false. 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.
Reduce Code Size for Custom Tags	reduce_tag_code <init-param>	Set this boolean to true for further reduction in the size of generated code for custom tag usage. The default is false.
(None)	req_time_introspection <init-param>	Set this boolean to true to enable 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. The default is false. 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 runtime. In this case, if req_time_introspection is enabled, the Web container will delay introspection until request-time. 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" /> <% } %>
(None)	setproperty_onerr_continue <init-param>	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="*". The default is false.
Generate Static Text as Bytes	static_text_in_chars <init-param>	Set this boolean to true to instruct the JSP translator to generate static text in JSP pages as characters instead of bytes. The default is false. 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"); %> The false default setting improves performance in outputting static text blocks.
(None)	jsp-print-null <init-param>	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.

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

<table>
<tr><th>Application Server Control Console JSP Property</th><th>XML Parameter</th><th>Description</th></tr>
<tr><td>Tags Reuse Default</td><td><code>tags_reuse_default</code> <code><init-param></code></td><td>Use this parameter to specify the mode for tag handler reuse, also known as tag pooling.
Set to <code>compiletime</code> to enable the compile-time model of tag handler reuse in its basic mode. This is the default value.
Set to <code>compiletime_with_release</code> to enable the compile-time model of tag handler reuse in its "with release" mode, where the tag handler <code>release()</code> method is called between usages of a given tag handler within a given page.
Set to <code>none</code> or <code>false</code> to disable tag handler reuse. You can override this value in any particular JSP page by setting the JSP page context attribute <code>oracle.jsp.tags.reuse</code> to a value of <code>true</code>.
Set to <code>runtime</code> 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 <code>oracle.jsp.tags.reuse</code> to a value of <code>false</code>.
Note that the <code>runtime</code> option and its equivalent, <code>true</code>, are deprecated in this release of OC4J.</td></tr>
<tr><td>JSP Page Timeout</td><td><code>jsp-timeout</code> attribute of the root <code><orion-web-app></code> element</td><td>Specify an integer value greater than <code>0</code> indicating the amount of time, in seconds, after which a JSP page will be removed from memory unless it is requested. The default is <code>0</code>.</td></tr>
</table>

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

Application Server Control Console JSP Property	XML Parameter	Description
Persistent TLD Caching	`jsp-cache-tlds` attribute of the root `<orion-web-app>` element	OC4J provides a persistent caching feature for TLD files, with a global cache for TLD files in well-known tag library locations, as well as an application-level cache for any application that uses TLD caching. See "Enabling the TLD Caching Feature" on page 7-27 for more information. By default, applications inherit the TLD caching value set at the global Web application level. However, if TLD caching is set at the application level, this value will override the global setting. ■ Set to `standard` (default) to search the Web application's `/WEB-INF` directory for files with the `.tld` extension. Any TLD files found will be added to the list of TLD files inherited from the global Web application. Note that the `/WEB-INF/lib` and `/WEB-INF/classes` directories will not be searched for files with the `.tld` extension. This is the default value set at both the global and application levels as of release 10*g* Release 3 (10.1.3). ■ Set to `on` to search all directories within the Web application for TLD files. Any found will be added to the list of TLD files inherited from the global Web application. ■ Set to `off` to disable persistent TLD caching. The `/WEB-INF/lib` and `/WEB-INF/classes` directories will not be searched for files with the `.tld` extension.. Note that the value of this parameter has implications on the ability to set multiple well-known tag libraries, as described in the Tag Lib Locations (`jsp-taglib-locations`) notation below.
Tag Libraries Location List	`jsp-taglib-locations` attribute of the root `<orion-web-app>` element	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 ***well-known tag libraries*** - directories in which you can place tag library JAR files to allow the libraries to be shared across multiple Web applications. See "Specifying Well-Known Tag Library Locations" on page 7-26 for more information. If TLD caching is disabled (`off`), the well-known tag library location is limited to a single directory, which is `ORACLE_HOME/j2ee/home/jsp/lib/taglib` by default. If TLD caching is enabled (`standard` or `on`), you can specify one or more well-known tag librarylocations using a semicolon-delimited list of directory paths. Note that the `jsp-taglib-locations` attribute can be set only in `global-web-application.xml`, not in an individual Web module's `orion-web.xml` file. Otherwise, the locations will be ignored. Note that tag libraries placed in a well-known directory must be packaged in a JAR file.

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

Application Server Control Console JSP Property	XML Parameter	Description
(None)	`simple-jsp-mapping` attribute of the root `<orion-web-app>` element	Set 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 `true`. If you must map the *.jsp extension to another servlet, set this parameter to `false`.

Setting JSP Parameters in Application Server Control Console

Oracle Enterprise Manager 10*g* Application Server Control Console is the administration console for an Oracle Application Server instance. It is installed by default with OC4J.

After logging in, access the JSP Container Properties page through **Administration**->**JSP Properties** links.

See the following for instructions on accessing the console:

- Accessing Application Server Control Console in Standalone OC4J
- Accessing Application Server Control Console in Oracle Application Server

Accessing Application Server Control Console in Standalone OC4J

The Application Server Control Console is installed and configured automatically when you install the OC4J software. It is started by default when OC4J is started.

The console is accessed through the `default` Web site, which is configured to listen for HTTP requests on port 8888. To access the console, simply type the following URL in a Web browser:

```
http://<hostname>:8888/em
```

Accessing Application Server Control Console in Oracle Application Server

The Application Server Control Console is installed and configured automatically when you install OC4J using the Oracle Universal Installer.

The console is started with all other installed Oracle Application Server components using the OPMN command-line tool, `opmnctl`, which is installed in the `ORACLE_HOME/opmn/bin` directory on each server node. Start all installed components by issuing the following command:

```
cd ORACLE_HOME/opmn/bin
opmnctl startall
```

In a typical Oracle Application Server installation, all Web applications, including Application Server Control Console, are accessed through Oracle HTTP Server (OHS). Use the following URL to access the console:

```
http://<ohs_host_address>:<port>/em
```

- `<ohs_host_address>` is the address of the OHS host machine.; for example, `server07.company.com`

- `<port>` is an HTTP listener port assigned to OHS by OPMN. Run the following `opmnctl` command on the OHS host machine to get the list of assigned listener ports from OPMN:

  ```
  opmnctl status -l
  ```

  Supply the port designated as `http1` in the OPMN status output as the value for `<port>`:

  ```
  HTTP_Server | HTTP_Server | 6412 | Alive | 1970872013 | 1
  6396 | 0:48:01 | https1:4443,http2:722,http1:7779
  ```

Setting JSP Parameters in the XML Configuration Files

In an OC4J development environment, you can set JSP configuration parameters directly in the global and module-specific configuration files.

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

Setting Servlet Initialization Paramters

An instance of the front-end servlet class - `oracle.jsp.runtimev2.JspServlet` - is created for each Web module instantiated within an OC4J instance. Note that this servlet class is the only servlet supported by OC4J.

The default parameters used to initialize each servlet instance are specified in `<init-param>` subelements of the the `<servlet>` element within the `global-web-application.xml` configuration file.

Note that you can override any of the default servlet parameters at the Web application level by specifying corresponding `<init-param>` elements in the J2EE standard `web.xml` deployment descriptor installed with the Web module. The default location for this file is the `ORACLE_HOME/home/j2ee/<appName>/<webModuleName>/WEB-INF` directory.

The following example illustrates how to set `<init-param>` elements containing servlet initialization paramaters within the `<servlet>` element 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>
```

Setting JSP Configuration Parameters

The parameters that control such functionality as JSP timeout and whether TLD caching is enabled are set as attributes of the root `<orion-web-app>` element in either `global-web-application.xml` or a Web module's `orion-web.xml` file. As with servlet initialization parameters, attributes set in `orion-web.xml` override the corresponding global definition in `global-web-application.xml`. The file for a deployed Web module is located in the `ORACLE_HOME/j2ee/home/application-deployments/<appName>/<webModuleName>` directory by default.

The following example shows how to set attributes of the root `<orion-web-app>` element within an XML file:

```
<orion-web-app development="false" jsp-timeout="30" ... >
 ...
</orion-web-app>
```

Configuring JSP Compilation in OC4J

The Java compiler can be invoked to execute *in-process* - within in the same JVM process as OC4J - or *out-of-process*, within in a separate JVM process.

By default, OC4J uses out-of-process compilation, and the compiler is invoked as a separate executable. The default compiler used is `javac` from the Sun Microsystems JDK.

You can configure OC4J to use a different compiler, or to compile in-process, by adding a `<java-compiler>` element with desired settings to the OC4J `server.xml` file. See the *Oracle Containers for J2EE Configuration and Administration Guide* for details.

Configuring Runtime JSP Retranslation and Reloading in OC4J

OC4J allows new JavaServer Pages (JSPs) to be added to an actively running Web module - as well as existing JSPs to be modified - without requiring an application redeployment or restart.

To use this feature, simply drop a new or updated JSP into the appropriate directory within the exploded WAR file structure in the OC4J instance, which is `ORACLE_HOME/j2ee/<instance>/applications/<appName>/<webModuleName>/`. OC4J will translate the page and load (or reload) it into the runtime.

This feature is controlled by the `main_mode` servlet initialization parameter. Possible settings are:

- `recompile` (default) to retranslate JSP pages that have changed
- `reload` to reload classes that were generated by the Web container and have changed (such as page implementation classes)
- `justrun` to run without any retranslation or reloading, which may provide optimal performance in production environments

See "Summary of JSP Configuration Parameters" on page 3-1 for additional information on the `main_mode` parameter.

Notes:

- 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 Containers for J2EE Servlet Developer's Guide*.

Key JSP-Related Support Files Provided with OC4J

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

- `ojsp.jar`: classes for the Web 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
- `ojdbc14.jar` / `classes12.jar` / `classes111.jar`: for the Oracle JDBC drivers (for JDK 1.4, 1.2 or higher, respectively)
- `runtime12.jar` / `runtime12ee.jar` / `runtime11.jar` / `runtime.jar` / `runtime-nonoracle.jar`: for the Oracle SQLJ runtime (for JDK 1.2.x or higher with Oracle9*i* or higher JDBC, JDK 1.2.x or higher enterprise edition with Oracle9*i* or higher JDBC, or any JDK environment with non-Oracle JDBC drivers, respectively)
- `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)

4 Precompiling JSPs with ojspc

This chapter describes the JSP pretranslation and precompilation capabilities provided by the ojspc utility packaged with OC4J. The following sections discuss ojspc functionality:

- How the ojspc Utility Works
- Using ojspc
- Precompiling One or More JSPs
- Precompiling JSPs within a WAR File
- Complete Summary of ojspc Command Line Options

How the ojspc Utility Works

This chapter describes the basic precompliation functionality provided by ojspc, as well as batch precompilation of JSPs with an archive file. It includes the following sections:

- Overview of Basic ojspc Functionality
- Overview of Batch Pretranslation of WAR Files

Overview of Basic ojspc Functionality

For a simple JSP page, default functionality for ojspc is as follows:

- Invokes the JSP translator to translate the JSP file into Java page implementation class code, producing a .java file.
- Invokes the Java compiler to compile the .java file, which produces a .class file for the page implementation class.

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 the current working directory from which you ran ojspc .

Output includes the following 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

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.

Overview of Batch Pretranslation of WAR Files

The `ojspc` utility accepts not only JSP files for translation, but can also accept WAR or EAR files for ***batch pretranslation***. The resulting `.class` files and any Java resource files are output to a nested JAR file inside the WAR file. The nested JAR file is set in the `WEB-INF\lib` path within the WAR, enabling the archive to be deployed as-is into OC4J.

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.
3. Tag files within the archive are also compiled if they used in a JSP within the archive.
4. Updates the archive with a new nested JAR file on the `WEB-INF\lib` path, and adds the resulting `.class` files and any Java resource files into this file. Any `.java` files that were created in the process are discarded.

The name of the nested JAR file includes the base name of the resulting archive file and has the `.jar` extension. For example, if `ojspc` is run on `sample.war`, the nested JAR file name within the WAR would be `__oracle_jsp_sample.jar`.

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. The `.class` and resource files in the nested JAR file are located in the same directory path as would be the case if the original JSP files were translated after extraction.

Using ojspc

The `ojspc` utility can be invoked from any directory, provided the location of the `ojspc` executable is added to the PATH on the host machine. The general `ojspc` command-line syntax is as follows:

```
ojspc [option_settings] file_list
```

The source file list can include JSP files and other source files (`.java`), or archive files (JAR, WAR, EAR, or ZIP files). To precompile all of the JSP files within a directory, specify `*.jsp` as the value for `file_list`.

See the following sections for specific `ojspc` usage scenarios:

- Precompiling One or More JSPs
- Precompiling JSPs within a WAR File

Important:

- The location of `ojspc.bat` must be added to the PATH to enable `ojspc` to be invoked from any working directory. The default path to `ojspc.bat` is:

  ```
  ORACLE_HOME/j2ee/home/jsp/bin/ojspc
  ```

The following are the most commonly used `ojspc` options used to control file generation and placement. See "Complete Summary of ojspc Command Line Options" on page 4-4 for the complete list of options.

- `-appRoot` to specify an application root directory if the directory containing the JSP source file(s) is different from the working directory in which `ojspc` is being run
- `-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 (`.res`) file.

Precompiling One or More JSPs

This section provides usage scenarios for precompiling JSPs.

Precompiling a Single JSP

This example illustrates basic `ojspc` usage, which translates and compiles the specified JSP.

```
cd /source
ojspc index.jsp
```

In this case, the following files will be generated within the `/source` working directory:

```
_index.class
_index.java
```

Precompiling Multiple JSPs

In the next example, all of the JSPs within the `myapp/mysrcdir/` are compiled:

```
cd /source
ojspc myapp/mysrcdir/*.jsp
```

By default, the resulting `.java` and `.class` files will be generated in a new subdirectory named `_myapp/_mysrcdir` within the `/source` working directory.

Precompiling JSPs in a Different Source Directory

Now presume that you run `ojspc` from a directory outside of the directory structure containing the source files. In this case, you must specify the application root using the `-appRoot` option, in addition to the absolute path to the files.

```
cd /stuff
ojspc -appRoot /source D:/source/myapp/mysrcdir/*.jsp
```

The resulting files will be generated in `/stuff/_source/_myapp/_mysrcdir`.

Generating Different File Types in Different Directories

The next example will generate Java class and files in the directory specified with the `-dir` option, and `.java` source files in the directory specified using `-srcdir`. The

-extres option will place the static text in the JSP into a Java resource file named _MyPage.res, which is useful when a page contains a lot of static content.

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

Be aware of the following when using ojspc:

- Use spaces between JSP file names in the file list to compile specific JSPs. Note that the path must be specified for each JSP if the files are not in the working directory.
- Specify *.jsp to compile all JSPs within the target directory.
- 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.

Precompiling JSPs within a WAR File

You can run ojspc on a WAR file, or on an EAR containing one or more nested WAR files, and precompile the JSPs within. The utility will create a JAR named __oracle_jsp_<warFileName>.jar containing the compiled Java classes on the WEB-INF\lib path within the WAR. The resulting WAR can then be deployed as-is.

For example, assume that ojspc is run on sample.ear, which includes a sample-web.war containing several JSPs. Note that because no output file is specified, the existing WAR will be updated with the new JAR file.

```
ojspc sample.ear
```

The updated sample-web.war will now include the following file on the WEB-INF\lib path:

```
__oracle_jsp_sample-web.jar
```

Additional usage notes:

- By default, ojspc updates the original archive file with the new JAR. If you want keep the original archive file unaltered, use the -output option to specify a new archive file name.
- 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).
- Use the -deleteSource option if you do not want processed source files to appear in the resulting archive file. Be sure to read the description of this option under "Complete Summary of ojspc Command Line Options" on page 4-4 before using it.
- Any .java files created during translation are discarded from the updated WAR.

Complete Summary of ojspc Command Line Options

Table 4–1 below summarizes the command-line options supported by the ojspc pretranslation utility.

Table 4–1 Options for ojspc Pretranslation Utility

Option	Description
`-addclasspath`	Use to specify additional classpath entries for `javac` to use when compiling generated page implementation class source.

`-approot`	Use to specify an application root directory. This option is used only when `ojspc` is run from a different directory other that containing the files to be translated. The default is the `ojspc` working directory.
	
	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.
	
	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:
	`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.

Table 4–1 (Cont.) Options for ojspc Pretranslation Utility

Option	Description

`-batchMask`	Use to specify source files to process in an archive file during batch pretranslation. 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. Note that file masks specified in this option are *not* case-sensitive.
	
	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 `.jsph` files:
	
	`ojspc -batchMask "*.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" myapp.ear`

`-debug`	Include to generate SMAP debugging data within the generated Java class file. An optional setting, `file`, generates debug information in an SMAP file within the working directory. The filename is the same as the generated Java class, with the .smap extension. For example, the debug data for `foo.jsp` will be output to `_foo_jsp.smap`.

Table 4–1 (Cont.) Options for ojspc Pretranslation Utility

Option	Description
-deleteSource	Include if you do not want processed source files to appear in the resulting archive file during batch pretranslation. This includes .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. 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.) Usage 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.
-dir (or -d)	Use 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. 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 ■ This option is ignored during batch pretranslation of WAR or EAR files. ■ In environments such as Windows and Unix that allow spaces in directory names, enclose the directory name in quotes.
-extend	Specify the class for the generated page implementation class to extend. Do not use this option for batch pretranslation.
-extraImports	Use to add imports beyond the default JSP packages imported by OC4J. Specify package names or fully qualified class names for any additional imports . 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`

Table 4–1 (Cont.) Options for ojspc Pretranslation Utility

Option	Description
`-extres`	Include to place static content of the page into a Java resource file instead of into the service method of the generated page implementation class. 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 "Managing Heavy Static Content or Tag Library Usage" on page 6-8. The resource file is placed in the same directory as output `.class` files. The 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.
`-forgiveDupDirAttr`	Include to avoid JSP translation errors if you have duplicate settings for the same directive attribute within a single JSP translation unit.
`-help` (or `-h`)	Displays `ojspc` usage information in the console.
`-ignoreErrors`	Include to force `ojspc` to continue processing if an error is encountered, unless the error is caused by an issue with one of the XML descriptors packaged with the JSPs, such as `web.xml` or `orion-web.xml`.
`-implement`	Specify an interface for the generated page implementation class to implement. Do not use this option for batch pretranslation.
`-noCompile`	Include to direct `ojspc` to *not* compile the generated page implementation class. This is in case you want to compile it later for some reason, such as with an alternative Java compiler.
`-noTldXmlValidate`	Include to *disable* XML validation of TLD files. By default, validation of TLD files is performed.
`-oldIncludeFromTop`	Include to specify that 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. This option provides backward compatibility with OC4J versions prior to Oracle9*i*AS Release 2.
`-output`	Use specify the name of the output archive file during batch pretranslation. All contents of the original archive file are copied into the specified archive file. The output `.class` files and any resource files from pretranslation are then 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 Batch Pretranslation of WAR Files" on page 4-2 for information about the nested JAR file.) Without the `-output` option, the original archive file is updated; no new archive file is created. The following is an example of `-output` usage: `ojspc -output myappout.war myapp.war`

Table 4–1 (Cont.) Options for ojspc Pretranslation Utility

<table>
<tr><th>Option</th><th>Description</th></tr>
<tr><td><code>-packageName</code></td><td>
Specify the package name for the generated page implementation class. If not specified, the package name is determined according to the location of the <code>.jsp</code> file relative to your current directory when you ran <code>ojspc</code>.
Consider an example where you run <code>ojspc</code> from the <code>/myapproot</code> directory, while the <code>.jsp</code> file is in the <code>/myapproot/src/jspsrc</code> directory (where <code>%</code> is a UNIX prompt):
<pre>% cd /myapproot
% ojspc -packageName myroot.mypackage src/jspsrc/Foo.jsp</pre>
This results in <code>myroot.mypackage</code> being used as the package name.
If this example did not use the <code>-packageName</code> option, the JSP translator (in its current implementation) would use <code>_src._jspsrc</code> as the package name by default. (Be aware that such implementation details are subject to change in future releases.)
</td></tr>
<tr><td><code>-reduceTagCode</code></td><td>
Include to direct further reduction in the size of generated code for custom tag usage. The default is <code>false</code>.
</td></tr>
<tr><td><code>-reqTimeIntrospection</code></td><td>
Include to allow 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. The default is <code>false</code>.
As a sample scenario for request-time introspection, assume a tag handler returns a generic <code>java.lang.Object</code> instance in the <code>VariableInfo</code> 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 <code>-reqTimeIntrospection</code> is enabled, the Web container will delay introspection until request-time.
An additional effect of this flag is to allow a bean to be declared twice, such as in different branches of an <code>if..then..else</code> loop. Consider the example that follows. Without <code>-reqTimeIntrospection</code> being enabled, this code would cause a parse exception. With it enabled, the code will work without error:
<pre><% if (cond) { %>
      <jsp:useBean id="foo" class="pkgA.Foo" />
<% } else { %>
      <jsp:useBean id="foo" class="pkgA.Foo2" />
<% } %></pre>
</td></tr>
<tr><td><code>-setpropertyOnErrContinue</code></td><td>
Include to continue iterating over request parameters and setting corresponding bean properties when an error is encountered during <code>jsp:setProperty</code> when property="*".
</td></tr>
<tr><td><code>-srcdir</code></td><td>
Use this option to place generated source files into a clean directory so that you conveniently know what files have been produced. Do not use this option for batch pretranslation.
Specify the location where <code>ojspc</code> will place the generated (<code>.java</code>) source files. The specified directory must already exist. The specified path is taken simply as a file system path, not an application-relative or page-relative path.
Subdirectories under the specified directory are created automatically, as appropriate, depending on the package.
The default is to use the current directory (your current directory when you executed <code>ojspc</code>).
</td></tr>
</table>

Table 4–1 (Cont.) Options for ojspc Pretranslation Utility

Option	Description
`-staticTextInChars`	Include to instruct the JSP translator to generate static text in JSP pages as characters instead of bytes. The default 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"); %>`
`-tagReuse`	Use this option to specify the mode of tag handler reuse (tag handler instance pooling). Values are as follows: ■ `runtime`: Enables 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`. Note that this option is deprecated in the current release. ■ `none`: Disables 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`. ■ `compiletime` (default): Enables the compile-time model of tag handler reuse in its basic mode. ■ `compiletime_with_release`: Enables 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.

`-verbose`	Include to direct `ojspc` to print status information 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.)
	`> ojspc -verbose myerror.jsp`
	`Translating file: myerror.jsp`
	` JSP files translated successfully.`
	`Compiling Java file: ./_myerror.java`

`-version`	Include to display the JSP version number.
`-xmlValidate`	Include to request XML validation of the `web.xml` file. By default, validation of `web.xml` is *not* performed.

5

Understanding JSP Translation in OC4J

This chapter describes the operation of the internal OC4J JSP translator, which translates JSP pages into Java servlet code. The following sections discuss general functionality of the JSP translator, focusing on its behavior in on-demand translation scenarios within the Oracle Application Server:

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

> **Important:** Implementation details in this section regarding package and class naming, file and directory naming, output file locations, and generated code are for illustrative purposes. The exact details are subject to change from release to release.

Features of Generated Code

The OC4J JSP translator generates standard Java code for a JSP page implementation class. This class is essentially a servlet class wrapped with features for JSP functionality.

This section discusses general features of the page implementation class code that is produced by the JSP translator from JSP source (typically `.jsp` or `.jspx` 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 Web 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.

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.

Notes:

- 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 "Managing Heavy Static Content or Tag Library Usage" on page 6-8. 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 `MyPage23.jsp` will always result in the string "`MyPage23`" being part of the page implementation class name, Java source file name, and class file name.

The base name is preceded by an underscore ("_"). Translating `MyPage23.jsp` results in the page implementation class `_MyPage23` in the source file `_MyPage23.java`, which is compiled into `_MyPage23.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/MyPage23.jsp`, for example, results in class `_MyPage23` 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 5-4.

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_foo2.jsp` results in `_My_2d_name__foo2` being the class name, in source file `_My_2d_name__foo2.java`. The hyphen is converted to a string of alpha-numeric characters. (An extra underscore is also inserted before "`foo2`".)

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 "`foo2`".

The generated source and class file names for a `.jspx` file are similar, except that _jspx is added to the name. For example, translating `MyPage.jspx` results in the source file `_MyPage_jspx.java`, which is compiled into `_MyPage_jspx.class`.

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 5-2.

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

This section describes files that are generated by the JSP translator and where they are placed in on-demand translation scenarios. (For precompilation scenarios, `ojspc` places files differently and has its own set of relevant options. See Chapter 4, "Precompiling JSPs with ojspc".)

Note: For information about general conventions used in naming output classes, packages, and files, see "General Conventions for Output Names" on page 5-2.

Files 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 by the JSP translator.

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:

- 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.

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

Oracle JSP Global Includes

The OC4J Web 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 Web 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.

Oracle global includes functionality pre-dates the JSP 2.0 specification. Now that this functionality has been folded into the specification, for portability it is strongly recommended that you use the JSP specification mechanism in all new development.

Also be advised that the Oracle global includes functionality may be deprecated in a future release.

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

- Global bean declarations (formerly supported through globals.jsa)
- Common page headers or footers
- Code with functionality that is equivalent to that of translate_params

Global Includes File and Examples

This section provides an overview of the ojsp-global-include.xml file as well as some examples.

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 "`/dir`". The setting can also include a slash after the directory name, such as "`/dir/`", 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='no'?>
<!DOCTYPE ojsp-global-include SYSTEM 'ojsp-global-include.dtd'>

<ojsp-global-include>
  <include file="/header.html">
     <into directory="/dir" />
  </include>
  <include file="/footer.html" position="bottom">
     <into directory="/dir" subdir="false" />
     <into directory="/dir/part/" subdir="false" />
  </include>
  <include file="/footer2.html" position="bottom">
     <into directory="/dir/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 `dir` 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 `footer.html` file is included at the bottom of any JSP page in the `dir` directory or its `part` 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="/footer.html" %>
  ```

- The `footer2.html` file is included at the bottom of any JSP page in the `part2` subdirectory of `dir`. 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=".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" %>
```

6

Working with JSP

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

The following sections are included:

- Before You Start
- General JSP Programming Strategies
- JSP Best Practices
- Working with Servlets
- Processing Runtime Errors

Before You Start

The following sections discuss some considerations you should be aware of before you begin coding or using JSP pages in the OC4J environment:

- Understanding Application Root Functionality
- Understanding OC4J Classpath Functionality
- Packages Imported By Default in OC4J
- Following JSP File Naming Conventions
- JDK1.4 Issue: Classes Not in Packages Cannot Be Invoked

Understanding 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 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/dir/abc.html`:

```
http://host:port/mybank/dir/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
```

Understanding OC4J Classpath Functionality

The OC4J Web 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 Web

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 "Generated Files and Locations" on page 5-4 for information.

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

Packages Imported By Default in OC4J

The OC4J Web container by default imports the following packages into any JSP page, in accordance with the JSP specification. No page directive import settings are required to use these packages in a JSP:

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

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

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

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

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

- Specify additional package names or fully qualified class names in one or more page directive import settings. For more information, see the page directive under "Directives" on page 1-5.

  For multiple pages, you can accomplish this through *global includes* functionality. See "Oracle JSP Global Includes" on page 5-5.

- 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:

  – "Summary of JSP Configuration Parameters" on page 3-1

  – "Complete Summary of ojspc Command Line Options" on page 4-4

JDK1.4 Issue: Classes Not in Packages Cannot Be Invoked

The Sun Microsystems JDK1.4 and JDK 1.5 ship with OC4J. As such, the following is something to consider if migrating from an earlier JDK.

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).

Notes:

- The `javac -source` compiler option is intended to allow JDK1.3. code to be processed seamlessly by the JDK1.4 compiler, but this option does not account for the "classes not in packages" issue.
- Only the JDK1.4 and JDK1.5 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 JDK1.4 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, v.4.0 and v.3".

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:

- Using JavaBeans Versus Scriptlets
- Using Static Includes Versus Dynamic Includes

Note: In addition to being aware of what is discussed in this section, you should be aware of JSP translation issues and behavior. See Chapter 5, "Understanding JSP Translation in OC4J".

Creating Traditional Versus Scriptless JSP

A major focus in JSP development has been on the creation of "scriptless" JSPs - pages that do not include embedded Java scripting elements such as scriptlets or runtime expressions. Scriptless JSPs offer a number of advantages over traditional script-based JSP development:

- Removal of JSP scriptlets and expressions from JSP pages
- Division of labor between JSP authors and Java developers. core philosophy of JSP has always been the division of labor and skills between JSP page authors, who are responsible for creating the HTML and JSP markup that comprises a JSP page, and Java developers, who are responsible for implementing the Java components that provide the processing logic.
- Cleaner, more readable JSP
- Ease of maintenance

The sample JSP code on page 1-4 is an example of a scriptless JSP.

Since JSP 1.1, page authors have been able to create largely Java-free pages that utilize standard action tags and custom tags to access the Java functionality provided through JavaBeans and tag handler instances. However, creating truly scriptless pages presented a number of challenges, including limits on data access and the complexity of creating custom tag handler classes.

The JSP 2.0 release dramatically simplifies scriptless page authoring with a number of major enhancements. Among these is the full integration of the expression language (EL) functionality into the JSP specification, giving the EL access to all JSP page context objects, variables and request parameters, as well as JavaBean properties and collection elements. With the EL, you can access and manipulate application data without having to use Java scriptlets or expressions. See "Simplified JSP Authoring with the Expression Language" on page 1-18 for more on the expression language.

It is also now much easier to create and use custom tags in your JSP pages. The JavaServer Pages Standard Tag Library (JSTL) provides a number of tag libraries that encapsulate much of the functionality most often needed by JSP authors. Creating custom tag handlers has been greatly simplified with the new SimpleTag interface. In fact, JSP authors can now create completely Java-free tag libraries using tag files, which are written completely in JSP syntax.

Note that JSP 2.0 provides backward compatibility with JSP 1.x, meaning that Java scripting elements can be used in pages written in JSP 2.0 syntax.

Using JavaBeans Versus Scriptlets

A key advantage of JavaServer Pages technology is the ability to separate the Java code containing the business logic and determining the dynamic content 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. 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.

Using Static Includes Versus Dynamic Includes

You have two options for including JSP pages in a JSP page.

The `include` directive, described in "Directives" on page 1-5, 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 JSP Action Tags" on page 1-11, 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.)

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 "Monitoring Your JSP Application" on page 6-7.)

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

Monitoring Your JSP Application

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.

```
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.

Managing Heavy Static Content or 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 JSP Action Tags" on page 1-11 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 "Summary of JSP Configuration Parameters" on page 3-1.

  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.

Using Method Variable Declarations Versus Member Variable Declarations

In "Scripting Elements" on page 1-6, 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 f=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 f=0.0;                  // *** f 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.

Working with Page Directives

This section discusses the following `page` directive characteristics:

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

Duplicate Settings of Page Directive Attributes Are Disallowed

The JSP specification states that a Web container must verify that directive attributes, with the exception of the `page` directive `import` attribute, are not re-set with different values within a single JSP translation unit (a JSP page plus anything it includes through `include` directives).

For backward compatibility to the JSP . standard, where duplicate settings of directive attributes are allowed, OC4J provides the `forgive_dup_dir_attr` configuration parameter. In JSP 2.0, you only need to set this parameter when the different attributes have different values. See "Summary of JSP Configuration Parameters" on page 3-1 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.

  This example is valid:

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

  or:

  ```
  ------------------------------
  <%@ page buffer="0kb" %>
  <%@ include file="b.jsp" %>
  ------------------------------
  ------------------------------
  b.jsp
  <%@ page session="false" %>
  ------------------------------
  ```

  However, this example would require that the `forgive_dup_dir_attr` parameter be set.

  ```
  <%@ page buffer="none" %>
  <%@ page buffer="0kb" %>
  ```

  or:

  ```
  <%@ page buffer="none" buffer="0kb" %>
  ```

  or:

  ```
  ------------------------------
  <%@ page buffer="0kb" %>
  <%@ 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 "Using Static Includes Versus Dynamic Includes" on page 6-6 for more information.

  Therefore, the following is okay:

  ```
  ------------------------------
  <%@ page buffer="0kb" %>
  <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.

Workarounds for the 64K Size Limit for Generated Methods

The Java Virtual Machine (JVM) limits the amount of code to 64K (65536 bytes) per Java method. If your application uses large JSPs, it is possible to exceed this limit during runtime. As a general rule, keep the file size of JSPs to a minimum.

If your JSP uses tag libraries heavily, enable the **Applications**->**JSP Container Properties**->**Reduce Code Size for Custom Tags** property (or the `reduce_tag_code` configuration parameter in `global-web-application.xml`) to reduce the size of generated code for custom tag usage. Note that this may impact JSP compilation performance.

Following 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 2.0 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.

Understanding JSP Preservation of White Space and Use with Binary Data

Web 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 : 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=5>
<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=5>
<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=5>
<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 20:9:20 PDT 2000


<B>Enter name:</B>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="user" SIZE=5>
<INPUT TYPE="submit" VALUE="Submit name">
</FORM>
</BODY>
</HTML>
```

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 Web 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 Web 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.

JSP Best Practices

The following sections discuss best practices to consider when developing JSP pages for deployment into OC4J.

Beware of HTTP Sessions

HTTP sessions add performance overhead to your Web applications due to the amount of memory used. Sessions are enabled in JSP by default.

Avoid Using HTTP Sessions If Not Required

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.

Always Invalidate Sessions When No Longer In Use

If your JSPs do use HTTP sessions, ensure that you explicitly cancel each session using the `javax.servlet.http.HttpSession.invalidate()` method to release the memory occupied.

The default session timeout for OC4J is 30 minutes. You can change this for a specific application by setting the `<session-timeout>` parameter in the `<session-config>` element of the application's `web.xml` file.

Pre-translate JSP Pages Using the ojspc Utility

You might consider using the `ojspc` utility to pretranslate JSP pages before deployment. This avoids the performance cost of translating pages as they are first

accessed by users. See Chapter 4, "Precompiling JSPs with ojspc" for details on using this utility.

Ensure Updated Objects Are Re-set on HTTP Sessions

When creating JSPs for distributable Web applications, ensure that updates to session objects are replicated in a clustered environment by coding your pages to re-set changed objects on the HTTP session.

OC4J will serialize session objects that are saved in the session; however, session objects are not re-serialized when changes are made to an object's data members, meaning that the updated session state will not be replicated. Note that this issue is not unique to JSP; for example, servlets must also re-set changed objects on the session.

Your JSPs should include scriplets that call `setAttribute()` on the `HttpSession` for each modifiable session attribute to ensure that the session state is replicated.

If using `<jsp:useBean>` tags to create session-scope beans, call `setAttribute()` to re-set updated beans on the session. Properties set on the bean when the bean is created are set in the session; however, updates to bean property values are not.

Un-Buffer 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.

Forward to JSP Pages Instead of Using Redirects

You can pass control from one JSP page to another using one of two options: Including a `<jsp:forward>` standard action tag or passing the redirect URL to `response.sendRedirect()` in a scriptlet.

The `<jsp:forward>` option is faster and more efficient. When you use this standard action, the forwarded target page is invoked internally by the JSP runtime, which continues to process the request. The browser is totally unaware that the forward has taken place, and the entire process appears to be seamless to the user.

When you use `sendRedirect()`, the browser actually has to make a new request to the redirected page. The URL shown in the browser is changed to the URL of the redirected page. In addition, all request scope objects are unavailable to the redirected page because redirect involves a new request.

Use a redirect only if you want the URL to reflect the actual page that is being executed in case the user wants to reload the page.

Hide JSP Pages from Direct Invocation to Limit Access

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 JSP-Timeout for Efficient Memory Utilization

Set the `jsp-timeout` attribute of the `<orion-web-app>` element to 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`, indicating no timeout.

The `<orion-web-app>` element is found in the OC4J `global-web-application.xml` and `orion-web.xml` files. Modify the `global-web-application.xml` file to apply the timeout to all applications in an OC4J instance. To set configuration values to a specific application, set the file in the application-specific `orion-web.xml` file.

Package JSP Files In EAR File For Deployment

OC4J supports deployment of JSP pages by copying the files directly to the appropriate location. This is very useful when developing and testing the pages.

However, this practice is not recommended for releasing your JSP-based application for production. Always package JSP files in an Enterprise Archive (EAR) file to allow deployment in a standard manner and to allow deployment across multiple application servers.

Working with Servlets

Although coding JSP pages is convenient in many ways, some situations call for servlets. One example is when you are outputting binary data.

As such, 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:

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

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 JSP Action Tags" on page 1-11.) 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 JSP Action Tags" on page 1-11.

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 6-18.

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", "234"); %>
<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>
```

Processing Runtime Errors

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 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 1-5.)

See the Sun Microsystems *Java Servlet Specification, Version 2.4* 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 1-9.

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

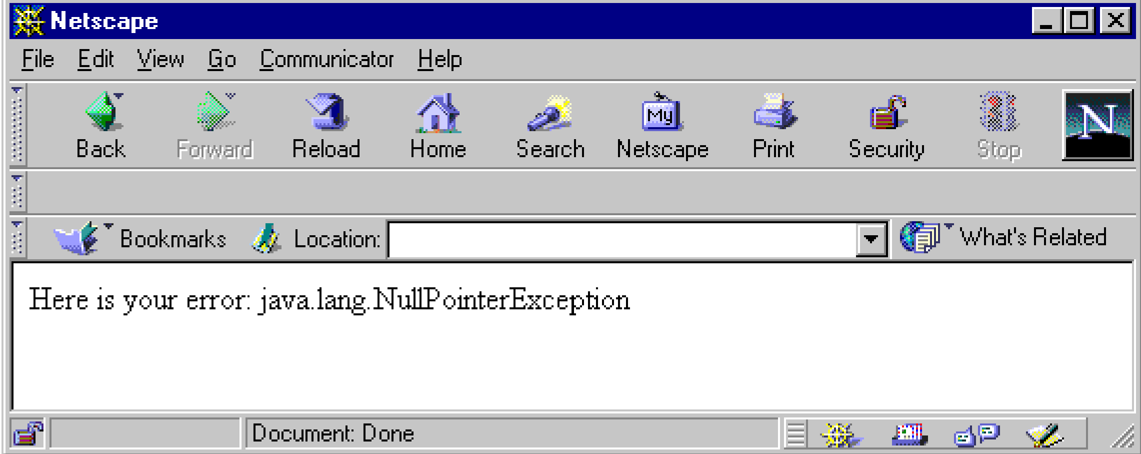
```
<HTML>
<BODY>
<%@ page errorPage="myerror.jsp" %>
Null pointer is generated below:
<%
   String s=null;
```

```
   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:



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.

7

Working with Custom Tags

This chapter discusses custom tags and 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:

- What Are Custom Tags?
- Working with Tag Handlers
- OC4J Tag Handler Features
- Working with Tag Files
- Sharing Tag Libraries Across Web Applications

The chapter offers a detailed overview of 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 Containers for J2EE JSP Tag Libraries and Utilities Reference*.

A multitude of resources are available for learning how to write and implement custom tags. For information beyond what is provided here, refer to the following resources:

- *JavaServer Pages Specification, version 2.0* from Sun Microsystems
- Sun Microsystems Javadoc for the `javax.servlet.jsp.tagext` package, at the following Web site:

```
http://java.sun.com/j2ee/1.4/docs/api/javax/servlet/jsp/tagext
/package-summary.html
```

What Are Custom Tags?

Custom tags, also known as *tag extensions*, are JSP elements that allow custom logic and output provided by other Java components to be inserted into JSP pages. The logic provided through a custom tag is implemented by a Java object known as a *tag handler*. When OC4J encounters a custom tag in a JSP during translation, it generates code to obtain and interact with the tag handler.

Custom tags are included in a JSP page using XML syntax. Tags may or may not contain a body. Tags can also contain XML attributes that match properties in the corresponding tag handler.

With the advent of JSP 2.0, you now have two options for creating custom tags:

- Tag handlers

Tags that require the creation of tag handler classes come in two types: Classic and simple.

Classic tag handlers have been available since JSP 1.1. Classic tags are considered somewhat cumbersome to write, in part because of the complexity of the Java interfaces used to implement each tag's corresponding tag handler class. They are also dependent on Java expressions for dynamic attribute values. However, these tag handlers are the only option if Java scriplets or expressions must be used in the tag body

Simple tag handlers are new in JSP 2.0, and offer a much simpler lifecycle and interface than classic tag handlers. Tag bodies accept JSP expression language (EL) expressions, allowing completely script-free tag development.

- Tag files

  Also new in JSP 2.0, *tag files* are revolutionary in that they allow tag libraries to be implemented completely in JSP or XML syntax, without the need to create and compile tag handler classes. Instead, tag files are translated into simple tag handlers by the OC4J JSP container and then compiled. Because of their ease of implementation, tag files offer an attractive alternative to writing tag handlers.

Related tag handlers or tag files can be packaged together in a *tag library*. Libraries developed as tag handlers must include an XML document known as a *tag library descriptor (TLD)* that describes the syntax of each tag and maps the tag to its corresponding handler class. Tag file libraries do not technically require a TLD, although descriptors are required if the library will be packaged as an archive for deployment.

The types of custom tags you create will depend on your needs. Tags that use either a classic or a simple tag handler are ideal when the flexibility of the Java language is required. Tag files are useful when creating tags that are presentation-centric or that take advantage of existing tag libraries, such as the JSTL.

Available Tag Libraries

Oracle provides extensive tag libraries with OC4J and other components. See the *Oracle Containers for J2EE JSP Tag Libraries and Utilities Reference* for details on available Oracle tag libraries.

Additional tag libraries are available from vendors and organizations. For example, the Jakarta Taglibs includes a wide selection of libraries available for download from `http://jakarta.apache.org/taglibs/`.

Note: The JSTL distribution is no longer installed in the `ORACLE_HOME/j2ee/home/jsp/lib/taglib` directory within OC4J.

See "Sharing Tag Libraries Across Web Applications" on page 7-24 for instructions on sharing tag libraries across deployed Web applications.

When Should You Consider Creating/Using Custom Tag Libraries?

Custom tags offer a number of benefits, including cleaner JSP code and code reusability. Some situations make creating and/or using custom tags almost mandatory. 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.

- Convenient JSP programming access to functionality that would otherwise require the use of a Java API is needed.
- Special manipulation or redirection of JSP output is required.

Eliminating Extensive Java Logic

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.

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. Similarly, while Oracle Application Server Personalization provides a Java API, OC4J also provides a tag library that you can use instead if you want to program a personalization application.

Manipulating or Redirecting JSP Output

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

For example, you can create a custom tag - `<cust:log>` - that will redirect the tag text into a log file instead of to a browser:

```
<cust:log>
   Text to log.
   More text to log.
   Yet more text to log.
</cust:log>
```

Working with Tag Handlers

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

- What Are Classic Tag Handlers?
- What Are Simple Tag Handlers?
- Attribute Handling and Conversions from String Values in Tag Handlers
- Implementing a Tag Handler
- Disabling or Enabling Tag Handler Reuse (Tag Pooling)
- Tag Handler Code Generation

What Are Classic Tag Handlers?

A classic tag handler is an instance of a Java class that directly or indirectly implements the standard `javax.servlet.jsp.tagext.Tag` interface. Although often regarded as somewhat complicated to write, classic tag handlers are your only option if Java scriplets or expressions must be used in the tag body or attribute values.

Classic Tag Handler Interfaces

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 classic 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.

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.

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 Tag Handler Features" on page 7-16 for information about the Oracle implementation.

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 attr="...", attr2="..." />
   ```

   This is equivalent to the following, which is also permissible:

   ```
   <oracust:mytag attr="...", attr2="..." ></oracust:abcdef>
   ```

   In this case, the tag handler should implement the `Tag` interface or extend `TagSupport`.

   The `<body-content>` setting for this tag in the TLD file should be `empty`.

2. There is a body; access to 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 attr="...", 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.

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: In most cases, you should 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.

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.

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.

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.

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.

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.

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.

What Are Simple Tag Handlers?

A new feature of JSP 2.0, simple tag handlers provide as much power as classic tag handlers, but are much easier to implement due to a much simpler Java interface and a more straightforward lifecycle.

Unlike classic tag handlers, simple tag handler objects are never cached and reused by the OC4J JSP container. Instead, an object is instantiated, executed and then discarded. There are no complicated caching semantics when using this interface since nothing is cached or reused. This simplified lifecycle helps make writing tag handlers easier and less prone to error.

Note that while this discussion focuses on implementing simple tag handlers as Java classes, simple tags can also be implemented completely in JSP syntax using tag files. See "Working with Tag Files" on page 7-18 for details. However, this option is only viable if the flexibility of Java is not required.

The SimpleTag Interface

Simple tag handler classes implement a single interface, `javax.servlet.jsp.tagext.SimpleTag`. The interface includes only one lifecycle method, `doTag()`. All iteration, body evaluation and other tag processing is performed within this method.

Ideally, your simple tag handler classes should extend the `javax.servlet.jsp.tagext.SimpleTagSupport` utility class, which implements the `SimpleTag` interface and provides a default implementation for the interface's methods. For example, this class provides `getJspBody()`, which returns the tag body passed to the handler.

Evaluation of the tag body is performed by the `setJspBody()` method, which is invoked by the OC4J JSP container with a `JspFragment` object encapsulating the body of the tag invocation. (See "Using JSP Fragments" on page 7-21 for a discussion on JSP fragments.) The tag handler can call the `invoke()` method on the `JspFragment` to evaluate the body as many times as needed. This ability to be invoked and evaluated multiple times is a key feature of fragments.

Unlike classic tags, simple tag extensions do not rely on the `PageContext` of the calling JSP page's underlying servlet, but instead rely on `JspContext`, which `PageContext` now extends. `JspContext` provides generic services such as storing the `JspWriter` and keeping track of scoped attributes, while PageContext has functionality specific to serving JSPs in the context of servlets. Through these objects, a simple tag handler can retrieve the other implicit objects (`request`, `session` and `application`) available from a JSP page.

Using Attributes

As with classic tags, the behavior of simple tags can be controlled through attributes that correspond to properties of the tag handler class.

There are three types of attributes: simple, fragment and dynamic.

Simple attributes are first evaluated by the OC4J JSP container when the tag is invoked, before being passed to the simple tag handler. Attributes are defined within the start tag using the syntax `attr="value"`. You can set the value as a `String` constant or an EL expression.

You can also define an attribute value within the body of a custom tag using the `<jsp:attribute>` element, a new element introduced in JSP 2.0. The following example uses `<jsp:attribute>` to use the output of the `<my:helloWorld>` custom tag to set the value of the `bar` property in a bean object:

```
<jsp:setProperty name="foo" property="bar">
   <jsp:attribute name="value">
      <my:helloWorld/>
   </jsp:attribute>
</jsp:setProperty>
```

Fragment attributes are used to create named fragments. See "Using JSP Fragments" on page 7-21 for an overview on JSP fragments.

As its name implies, a *dynamic attribute* is an attribute not specified in the tag definition. The ability of a tag to dynamically specify attributes is a new feature of JSP 2.0 that is applicable to both simple and classic tag handlers. Dynamic attributes are especially useful in tags that have a number of attributes that are all processed in a similar manner.

Tag handlers that support dynamic attributes must declare that they do so in the `tag` element of the TLD. For example:

```
<tag>
  <description>
   Tag that echoes all its attributes and body content
  </description>
  <name>echoAttributes</name>
  <tag-class>jsp2.examples.simpletag.EchoAttributesTag</tag-class>
  <body-content>empty</body-content>
  <dynamic-attributes>true</dynamic-attributes>
</tag>
```

Attribute Handling and Conversions from String Values in Tag Handlers

A tag handler class has an underlying property for each attribute of the custom tag. These properties are analogous to 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 Web 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 1-16.

- The second approach is where the attribute is a request-time attribute that is set using a request-time expression:

```
rtattr="<%=expression%>"
```

or:

```
rtattr="${ELexpression}"
```

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.

Using Scripting Variables in Tags

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" attr="..." attr2="..." />
```

This statement results in the object `myobj` being available to scripting elements in the page, according to the declared scope of `myobj`. 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 7-10.
- 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 7-11.

Generally, the more convenient `<variable>` mechanism will suffice.

The Web 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 7-11. The JSP1.2 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 7-9. 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 hard-coded 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.

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 Web 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/1.4/docs/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 Web 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:

- Variable name
- 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
- Variable scope

> **Important:** As of the OC4J 9.0.4 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.

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:bar attr="abc" >
   <foo:bar2 />
</foo:bar>
```

Within the code of the `bar2` tag handler class (class `Bar2Tag`, by convention), you can have a statement such as the following:

```
Tag bartag = TagSupport.findAncestorWithClass(this, BarTag.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 bar tag handler class (presumed to be BarTag in the example), as a java.lang.Class instance

The findAncestorWithClass() method returns an instance of the appropriate tag handler class, in this case BarTag, as a javax.servlet.jsp.tagext.Tag instance.

It is useful for a Bar2Tag instance to have access to the outer BarTag instance in case the Bar2Tag needs the value of a bar tag attribute or needs to call a method on the BarTag instance.

Implementing a Tag Handler

Creating either a classic or simple tag handler implemented in Java consists of the following key steps:

- Write and compile the tag handler class
- Package the tag handler in a tag library
- Define the tag in the tag library descriptor (TLD)

Several examples of simple tag handler implementations are provided with the sample applications that you can download from the Oracle Technology Network (OTN) Web site. After downloading and deploying the sample applications on OC4J, open the tag handler Java source files to see how each is implemented.

Creating the Tag Handler Class

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.

Note that a tag handler class must have a public no-arguments constructor.

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 Web 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.

Defining the Tag in the TLD

Each tag that your tag handler implements must be defined in the tag library descriptor file packaged with the tag library containing the handler class. The tag definition in the TLD fulfills two roles:

- Defines the syntax of the tag
- Provides the mapping between the tag and its corresponding tag handler class

The following snippet is based on the definition for the <my:shuffle> tag from shuffle.tld, the descriptor for the library containing the ShuffleSimpleTag handler class. This handler is included with the "Random Tic-Tac-Toe" sample application, which illustrates simple tag handler features. You can download this application from the Oracle Technology Network Web site.

The format for the TLD is shown in the following example. All TLD files are required to begin with a root <taglib> element that specifies the XML schema used to describe the TLD and the JSP version:

```
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-jsptaglibrary_2_0.xsd"
    version="2.0">
```

```
  <tlib-version>1.0</tlib-version>
  <short-name>SimpleTagLibrary</short-name>
  <uri>/SimpleTagLibrary</uri>
  <tag>
   ...
  </tag>
</taglib>
```

Next is the definition for the <my:shuffle> tag, contained within a <tag> element:

```
<tag>
  <name>shuffle</name>
   <tag-class>oracle.otnsamples.jsp20.simpletag.ShuffleSimpleTag</tag-class>
   <body-content>empty</body-content>
   <attribute>
     <name>fragment1</name>
     <required>true</required>
     <fragment>true</fragment>
   </attribute>
   <attribute>
     <name>fragment2</name>
     <required>true</required>
     <fragment>true</fragment>
   </attribute>
   <attribute>
     <name>fragment3</name>
     <required>true</required>
     <fragment>true</fragment>
   </attribute>
</tag>
```

The subelements of a <tag> element define the 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 "Working with Tag Handlers" on page 7-3 for information about tag handler classes.
- The <body-content> subelement indicates how the tag body (if any) should be processed.
- Each <variable> subelement (if any), with its further subelements, defines a scripting variable. See "Using Scripting Variables in Tags" on page 7-9 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 7-10 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 7-11 for more information.

- Each <attribute> subelement (if any), with its further subelements, provides information about a parameter that you can pass in when you use the custom tag - in this example, a named fragment. 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 10g (9.0.4), 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).

Declaring the Tag in a JSP Page

The tag library containing the tag handler is referenced in the file using a <taglib> directive. Note the prefix <my:...> which indicates that any tag including this prefix is defined in the corresponding TLD.

```
<%@ taglib uri="/WEB-INF/shuffle.tld" prefix="my" %>
```

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 1-23 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 Multiple Tag Libraries and TLD Files in a JAR File" which describes a scenario for packaging tag libraries to be shared by multiple Web applications.

Using the Tag in a JSP

The `<my:shuffle>` tag is used in the TictactoeManual.jsp page, which draws a "tic-tac-toe" table of three rows and three columns. Each column has a tile of a particular color, achieved by calling the `<my:tile>` tag with three different colored tiles, or images. The `<my:shuffle>` tag is used to wrap these calls. The tag handler randomly executes the three tile fragments, causing the colored tiles within each row to be shuffled each time the page is loaded.

```
<my:shuffle>
   <jsp:attribute name="fragment1">
     <tr>
       <my:shuffle>
         <jsp:attribute name="fragment1">
           <my:tile imageVal="../images/blue_plain.gif" />
          </jsp:attribute>
          <jsp:attribute name="fragment2">
            <my:tile imageVal="../images/yellow_plain.gif" />
       </jsp:attribute>
       <jsp:attribute name="fragment3">
         <my:tile imageVal="../images/pink_plain.gif" />
       </jsp:attribute>
         </my:shuffle>
      </tr>
    </jsp:attribute>
      ...
</my:shuffle>
```

OC4J 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 Tag Handler Reuse (Tag Pooling)
- Tag Handler Code Generation

Disabling or Enabling Tag Handler Reuse (Tag Pooling)

To improve performance, you can specify that tag instances be reused within each JSP page. This functionality is often referred to as *tag pooling*.

As of Oracle Containers for J2EE Release 3 (10.1.3), a *compile-time* tag pooling model is supported. 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).

Tag pooling in the OC4J JSP container is configured using the `tags_reuse_default` parameter. See "Summary of JSP Configuration Parameters" on page 3-1 for further information on setting this parameter.

> **Note:** The *runtime* tag pooling model - previously the default mechanism used - is deprecated in Oracle Containers for J2EE Release 3 (10.1.3).

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.

Notes:

- Remember to retranslate your JSP pages when switching between the `compiletime`, `compiletime_with_release` or `runtime` models 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`.

When Can the Compile-Time Tag Pooling Model Be Used?

In conformance with the JSP 2.0 specification, a tag instance can only be reused when the set of attributes defined for the tag is identical from usage to usage. This restriction is applicable to both the `compiletime` and `compiletime-with-release` tag pooling models.

The following example uses the tags from the JSTL `core` library to illustrate tag instance reuse.

```
<%@ taglib uri='http://java.sun.com/jstl/core' prefix='c' %>

// New instance of c:forEach tag is created.
// Note the inclusion of the "var" attribute.
<c:forEach var='item' begin='2' end='10'>

// New instance of c:out tag is created.
<c:out value="foo"/>

</c:forEach>

<%out.println("****");%>

// The first c:forEach tag instance cannot be reused, since the "var"
// attribute does not exist. A new tag instance is therefore created.
<c:forEach begin='1' end='10'>

// The existing c:out tag instance is reused, as
// the set of attributes is identical in both usages.
<c:out value="bar"/>
```

Code Pattern for the compiletime Tag Pooling Model

The following illustrates the code pattern for the default `compiletime` model for tag pooling.

```
try {
  tag01.doStartTag();
  ...
  tag01.doEndTag();
  // reuse tag01 without calling release()
  tag01.doStartTag();
  ...
  tag01.doEndTag();
}
catch (Throwable e) {
  tag01.release();
}
```

Code Pattern for the compiletime-with-release Tag Pooling Model

The following illustrates the code pattern for the `compiletime-with-release` option, which calls the `release()` method on the tag handler object between uses of the same tag handler within the same JSP.

```
try {
  tag01.setPageContext(pageContext);
  tag01.doStartTag();
  ...
  tag01.doEndTag();
  tag01.release();
  // reuse tag01 with calling release()
  tag01.doStartTag();
  ...
  tag01.doEndTag();
  tag01.release();
}
catch (Throwable e) {
  tag01.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 Tag Handler Reuse (Tag Pooling)" on page 7-16, the effect is not maximized when `reduce_tag_code` is also `true`.

Working with Tag Files

The following sections provide an overview and understanding of tag files:

- What Are Tag Files?
- Tag Body Processing
- Using Attributes in Tag Files

- Exposing Data through Variables in Tag Files
- Using JSP Fragments
- Implementing a Tag File

What Are Tag Files?

Introduced with JSP 2.0, tag files allow JSP authors to create custom tag libraries completely using JSP syntax, without requiring any knowledge of Java. In fact, writing a tag file is just like writing a JSP, with a few differences. Because of their ease of implementation, tag files offer an attractive alternative to traditional custom tag handlers.

The source for a tag file is simply a text file containing a reusable fragment of JSP code. Valid content that can be used in a tag file include:

- Standard JSP actions (such as `<jsp:useBean>`)
- Custom tags
- JSTL tags
- EL expressions or functions
- Template text

Unlike traditional custom tags, tag files do not require a corresponding tag handler class; instead each file is translated and compiled into a simple tag handler by the OC4J JSP container. Tag files effectively remove the need for Java programming, opening tag library creation to JSP authors who do not possess a strong working knowledge of Java.

Standard JSP directives are used in a tag file to provide processing instructions, just as in a JSP page. The one exception is the `page` directive, which cannot be used. Instead, tag files use the `tag` directive, which includes attributes specific to tag file processing.

Additional directives, such as `attribute` and `variable`, can only be used in tag files. "Using Attributes in Tag Files" on page 7-20 and "Exposing Data through Variables in Tag Files" on page 7-20 for details.

Tag Body Processing

How the body of a custom tag is processed is defined in the `body-content` attribute of the corresponding tag file's `tag` directive. The default value is `scriptless`, indicating that any of the JSP or static text elements listed above can be used in the tag body - although as the value infers, Java scripting elements are not allowed. Tags that do not accept a tag body should be declared as `empty`, as shown below:

```
<%@ tag body-content="empty" %>
```

The tag's body is evaluated by the `<jsp:doBody>` standard action, which can only be used within tag files. Any dynamic JSP elements in the tag body are called, and the output they produce is mixed with template text in the body in the evaluation result.

The result is saved in a variable, which can be named using either the `var` or `varReader` attributes of the `<jsp:doBody>` tag. The `var` attribute captures the result as a `String` and should be sufficient for most usage scenarios.

The `varReader` attribute is used to capture the result as a `java.io.Reader` object, which may be more efficient when combined with a standard or custom tag, or with

an EL function that reads its input from a `Reader`. If no variable is specified, the output is sent to the implicit `JspWriter` object.

As noted in the discussion of the SimpleTag interface, the body of the tag invocation is translated to a `JspFragment` object, which is then passed to the tag handler for processing. In this case, the value of `text` is displayed in bold and italic fonts in a Web browser. Fragments are very useful in the context of tag files, as discussed in "Using Attributes in Tag Files" below.

Using Attributes in Tag Files

Attributes are declared directly within tag files using `attribute` directives. In the example below, the `required` attribute is set to `true`, meaning that the page author must specify a value for this attribute when a tag element body passed in, or an error will be returned. The default value is false.

```
<%@ attribute name="category" required="true" %>
```

As with classic and simple tag handlers, it is also possible to use undeclared dynamic attributes in a tag file. This is done by setting the `dynamic-attributes` attribute of the tag directive to the name of a Map containing the names and values of the dynamic attributes passed during tag invocation. The Map must contain each dynamic attribute name and value as a key/value pair.

```
<%@ tag body-content="scriptless" dynamic-attributes="dynattrs" %>
```

Exposing Data through Variables in Tag Files

Variables are also declared directly within the tag file using the `variable` directive, which is analogous to the `<variable>` element used to declare variables used by a tag handler in a TLD. The name-given attribute specifies the variable's name and the variable-class attribute defines its type.

```
<%@ variable name-given="current" variable-class="java.lang.Object"
scope="NESTED" %>
```

As with tag handlers, the variable directive's scope attribute accepts one of three values, which control where the calling JSP sees the variable: AT_BEGIN, AT_END, or NESTED.

Tag files have access to a local page scope, which is distinct from the page scope of the calling JSP. The use of different scopes prevents confusion between the calling page and the tag file if they use the same names for page scope variables. In many cases, you may want to allow variable names to be specified by the calling JSP using an attribute supplied in a tag body, rather than hard-coding names in the tag file.

```
<%@ variable name-from-attribute="var" alias="current"
variable-class="java.lang.Object" scope="NESTED" %>
```

Here, the `name-from-attribute` and `alias` attributes are used instead of the `name-given` attribute used in the previous example. The `name-from-attribute` value is the name of the tag attribute providing the variable name.

The `alias` attribute value declares the name of the tag file's local page scope variable, which the OC4J JSP container copies to the calling JSP's page scope. This attribute is needed because the page author can assign any name for the variable in a custom tag, but a fixed name - the alias - must be used when developing the tag file.

Using JSP Fragments

A JSP fragment can be thought of as a template used to produce customized content. Introduced with JSP 2.0, JSP fragments are a new option for use with simple tag extensions, and as such are relevant for both tag files simple tag handlers. However, creating a fragment is typically done within the context of a tag file.

Like a tag file, a fragment is an encapsulation of dynamic JSP elements - such as standard JSP action tags, custom tags or EL expressions - and optionally static template text that is passed to a simple tag handler through a tag invocation.

A fragment can be invoked and evaluated by the tag handler zero or more times, as needed. Because the JSP elements within the fragment have access to the current values of all scoped variables, the result will typically be different from invocation to invocation. These qualities make fragments ideal for:

- Displaying conditional data or template text
- Displaying a set of data in an iteration

Creating a JSP Fragment

Creating a *named fragment* for use in a tag file can be thought of as a two step process: You must declare the fragment within the tag file, then define its body within a `<jsp:attribute>` standard action tag inside the body of a custom tag.

You declare the fragment in the tag file using an `attribute` directive. Setting the directive's `fragment` attribute to `true` will cause the fragment to be evaluated by the tag handler. The default setting of `false` forces the attribute to be evaluated by the OC4J JSP container *before* being passed to the tag handler; hence the fragment's ability to be evaluated "zero or more times".

```
<%@ attribute name="frag1" fragment="true" %>
<%@ attribute name="frag2" fragment="true" %>
```

You then define the associated logic within the tag file using JSP syntax. The following snippet from a tag file illustrates conditional logic implemented using JSTL tags. If the test evaluates true, `frag1` is invoked:

```
<c:when test="${empRow[2] >= 10000}">
   <c:set var="name" value="${empRow[0]}"/>
   <c:set var="phone" value="${empRow[1]}"/>
   <c:set var="salary" value="${empRow[2]}"/>
   <jsp:invoke fragment="frag1"/>
</c:when>
```

The fragment is invoked by name using the `<jsp:invoke>` standard action tag within the tag file. Like `<jsp:doBody>`, this action can only be used within tag files. The scoped variables within the tag body - in this case EL constructs - are set by the tag file, allowing the tag file code to customize the fragment each time it is invoked.

The fragment content, which as noted previously can consist of JSP elements and static HTML, is defined within the body of a `<jsp:attribute>` standard action tag within the body of a custom tag.

```
<tags:EmpDetails deptNum="80">
 <jsp:attribute name="fragment1">
   <tr bgcolor="#FFFF99" align="center">
    <td><font color="#CC0033">${name}</font></td>
    <td><font color="#CC0033">${phone}</font></td>
    <td><font color="#CC0033">${salary}</font></td>
   </tr>
```

```
  </jsp:attribute>
</tags:EmpDetails>
```

A Tag File Example

The following is a simple example illustrating the use of the `var` attribute to store the result of a tag body evaluation.

The sample `doBodyVarTest.tag` file defines a variable named `text` that will store the result of evaluating the body of the `<tags:doBodyVarTest>` tag in `Example.jsp`. It also defines two fragment attributes—`frag1` and `frag2`—that will store the result of an EL variable evaluation.

When a tag file is executed, the Web container passes it two types of fragments: fragment attributes and the tag body. Within the tag file, the `<jsp:invoke>` element is used to evaluate a fragment attribute, while the `<jsp:doBody>` element is used to evaluate a tag file body. The result of evaluating either type of fragment is sent to the response or is stored in an EL variable for later manipulation.

```
<%@ attribute name="frag1" required="true" fragment="true" %>
<%@ attribute name="frag2" required="true" fragment="true" %>
<%@ variable name-given="text" scope="NESTED" %>
 <jsp:doBody var="text" />
  <TABLE border="0">
   <TR>
    <TD>
     <b><jsp:invoke fragment="frag1"/></b>
    </TD>
   </TR>
   <TR>
    <TD>
     <i><jsp:invoke fragment="frag2"/></i>
    </TD>
   </TR>
  </TABLE>
```

The `Example.jsp` code creates an instance of the `doBodyVarTest` tag. Note the `<jsp:body>` tag, which contains a string that will be set by the tag handler into the `text` variable. The `<jsp:body>` tag is used to explicitly specify the body of a simple tag, and is the only option for specifying a tag body when one or more `<jsp:attribute>` elements appear in the body of a tag invocation.

```
<%@ taglib tagdir="/WEB-INF/tags" prefix="tags" %>

<html>
 <body>
  <tags:doBodyVarTest>
   <jsp:attribute name="frag1">
    ${text}
   </jsp:attribute>
   <jsp:attribute name="frag2">
    ${text}
   </jsp:attribute>
   <jsp:body>
    Have a great day!
   </jsp:body>
  </tags:doBodyVarTest>
 </body>
</html>
```

Implementing a Tag File

The process for implementing a tag library as tag files is straightforward: You need only to write the tag files as outlined in this chapter.

Creating the Tag File

As noted earlier in this chapter, writing a tag file is very similar to writing a JSP. See "What Are Tag Files?" on page 7-19 for a summary of elements that can be included in a tag file.

Be aware that tag files in your JAR file remain as uncompiled tag files, meaning that your code can be viewed by anyone with a text editor.

Packaging Tag Files

Tag files can be deployed in one of two ways:

- Drop the tags directly into a `/WEB-INF/tags` directory within an application's directory structure

  When deployed in this manner, tag files do not require a tag library descriptor. However, the tags will only be accessible to the single application, unless you copy the tags to the `/WEB-INF/tags` directory for each application.

- They can be packaged in a JAR file

  Packaging tag files in this manner requires a TLD. Any tag files within the JAR that are not defined in the TLD are ignored by the OC4J JSP container.

When packaged in a JAR, tag files are added under the `/META-INF/tags` within the JAR. The JAR will then be installed in the `/WEB-INF/lib/` directory of a Web application.

Each tag file in the JAR is defined in the TLD within a `<tag-file>` element, as opposed to the `<tag>` element used to define classic or simple tag handlers. Each `<tag-file>` element takes two sub-elements:

- The `<name>` element defines the tag name and must be unique
- The `<path>` element specifies the full path of the tag file and must therefore begin with `/META-INF/tags`

The example below shows a TLD defining two tag files packaged in a JAR for distribution:

```
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-jsptaglibrary_2_0.xsd"
   version="2.0">
  <tlib-version>1.0</tlib-version>
  <short-name>MyTagFiles</short-name
  <uri>/MyTagFiles</uri>
  <description>JSP 2.0 tag files</description>
  <tlib-version>1.0</tlib-version>
  <short-name>My Tag Files</short-name>
  <tag-file>
    <name>EmpDetails</name>
    <path>/META-INF/tags/mytags/EmpDetails.tag</path>
  </tag-file>
  <tag-file>
    <name>ProductDetails</name>
    <path>/META-INF/tags/mytags/ProductDetails.tag</path>
  </tag-file>
```

```
</taglib
```

Declaring the Tag File in a JSP

A JSP page imports a tag library implemented with tag files using the `taglib` directive. The tag library prefix is specified using the `prefix` attribute. However, the other attributes included vary depending on how the tag files are deployed.

If the tag files are not packaged in a JAR, the value of the `tagdir` attribute must be set to the context-relative path to the directory that contains the tag files.

```
<%@ taglib tagdir="/WEB-INF/tags/mytags" prefix="mytags" %>
```

If the tag files are packaged in a JAR, the `uri` attribute is used, and is set to the content of the `<uri>` element in the TLD.

```
<%@ taglib uri="/MyTagFiles" prefix="mytags" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

Sharing Tag Libraries Across Web Applications

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

- Packaging Multiple Tag Libraries and TLD Files in a JAR File
- Specifying Well-Known Tag Library Locations
- Enabling the TLD Caching Feature

Packaging Multiple Tag Libraries and TLD Files in a JAR File

The JSP specification allows the packaging of multiple tag libraries, and the TLD files that define them, in a single JAR file.

This section presents an example of multiple tag libraries packaged in a single JAR file. The JAR file includes tag handler classes, `tag-library-validator` (TLV) classes, and TLD files for multiple libraries.

The following lists the contents and structure of the JAR file. Note that in a JAR file with multiple TLD files, the TLD files must be located under the `/META-INF` directory or a subdirectory.

```
examples/BasicTagParent.class
examples/ExampleLoopTag.class
examples/BasicTagChild.class
examples/BasicTagTLV.class
examples/TagElemFilter.class
examples/TagFilter.class
examples/XMLViewTag.class
examples/XMLViewTagTLV.class
META-INF/xmlview.tld
META-INF/exampletag.tld
META-INF/basic.tld
META-INF/MANIFEST.MF
```

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 "Specifying Well-Known Tag Library Locations" on page 7-26. 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.

Key TLD File Entries

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; however, it must follow the XML namespace convention.I t 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.

The basic.tld file includes the following:

```
<taglib>

  <tlib-version>1.0</tlib-version>
  <jsp-version>2.0</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>

  <tlib-version>1.0</tlib-version>
  <jsp-version>2.0</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 Deployment Descriptor Entries

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.

The <taglib> element can include two subelements:

- <taglib-uri>

Contains the shortcut URI that will be used as the value of the `uri` attribute in the `taglib` directive in JSP pages that use the tag.

- `<taglib-location>`

  Contains the unique identifier for the 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.

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 Well-Known Tag Library Locations" on page 7-26 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 Multiple Tag Libraries and TLD Files in a JAR File" on page 7-24 for related information.

```
<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" %>
```

Specifying Well-Known Tag Library Locations

As an extension of the standard "well-known URI" functionality described in the JSP specification, OC4J supports the use of one or more directories, known as *well-known*

tag library locations, where you can place tag library JAR files that will be shared across multiple Web applications.

The default well-known tag library location is the `ORACLE_HOME/j2ee/home/jsp/lib/taglib/` directory. A tag library installed in this location will be available by default to all Web applications deployed to the OC4J instance.

You can also define additional shared tag library locations, and install tag library JAR files to be shared across applications in these directories. Defining a well-known tag library location is a two-step process:

1. Define each directory in the `jsp-taglib-locations` attribute of the of the `<orion-web-app>` element in the `ORACLE_HOME/j2ee/home/config/global-web-application.xml` file. Separate each location with a semicolon.
2. Add a `<library>` element for each directory to `ORACLE_HOME/j2ee/home/config/application.xml`, the configuration file for the `default` application. Set the `path` attribute to the directory containing the tag library JAR file.

Important: The ability to specify and utilize *multiple* shared tag library locations is determined by the value of the `jsp-cache-tlds` attribute of the `<orion-web-app>` element. See Table 7–1, " TLD Caching Parameters" on page 7-28 for details.

Enabling the TLD Caching Feature

As part of its support for sharing tag libraries, OC4J provides a persistent caching feature for TLD files. This includes 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 turn it off, however, if either of the following circumstances are true:

- Your application does not use tag libraries.
- You have pre-translated the JSP pages and none of the TLD files use `<listener>` elements for tag library event listeners.

TLD caching is enabled or disabled through the `jsp-cache-tlds` attribute of the `<orion-web-app>` element.

- At the global level, TLD caching is set through this attribute in the `global-web-application.xml` file, the Web configuration file for the global Web application. The value set in this file is the default inherited by all other Web applications deployed to the OC4J instance.
- At the application level, caching is set in the application-specific `orion-web.xml` file. The setting in this file overrides the default setting in `global-web-application.xml`.

The following table summarizes the values for the `jsp-cache-tlds` attribute.

Table 7–1 TLD Caching Parameters

Value of jsp-cache-tlds	Value set in global-web-application.xml
standard	TLD caching is enabled. This is the default setting at the global and application level. Add tag library JARs to the default well-known tag library location, which is ORACLE_HOME/j2ee/home/jsp/lib/taglib. The tag libraries will be available to all Web applications. Note that TLD (*.tld) files must be placed in the /WEB-INF directory. Do NOT put TLD files in the /classes or /lib subdirectories.
on	TLD caching is enabled. Add JAR files containing tag libraries to one of the following: ■ The default well-known tag library location, which is ORACLE_HOME/j2ee/home/jsp/lib/taglib. ■ An additional location specified in the jsp-taglib-locations attribute of the <orion-web-app> element in global-web-application.xml. See "Specifying Well-Known Tag Library Locations" on page 7-26 for details on specifying additional locations.
off	TLD caching is disabled. Add tag library JARs to the default well-known tag library location only, which is ORACLE_HOME/j2ee/home/jsp/lib/taglib. A different location can be specified as the value of the well_known_taglib_loc initialization parameter in ORACLE_HOME/j2ee/home/config/global-web-application.xml. Initialization parameters are specified within <init-param> subelements of <servlet>jsp</servlet> notation in this file. See "Setting JSP Parameters in the XML Configuration Files" on page 3-8 for details.

Important:

- 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.

Understanding the TLD Cache Features and Files

For any application that uses TLD caching, whether it is enabled at the global or 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`. If the `jsp-cache-tlds` attribute of the `<orion-web-app>` element is set to `standard`, TLDs must NOT be placed in either `/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.
- 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:

- A file, `_GlobalTldCache`, that contains resource information, as described above, for any well-known locations.
- 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`. 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`.

  If the `jsp-cache-tlds` attribute of the `<orion-web-app>` element is set to `standard`, TLDs must NOT be placed in either `/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. The default level is 3.

8

Understanding JSP XML Support in OC4J

JavaServer Pages technology is increasingly seen as an effective model for producing XML documents. New enhancements released with JSP 2.0, including an improved XML syntax, make JSP even more complementary to XML technology and more accessible to XML tools.

The OC4J JSP container fully implements JavaServer Pages support for XML as described in the JavaServer Pages 2.0 specification. This includes support for:

- XML-style equivalents to JSP syntactical elements
- The concept of the "XML view" of a JSP page
- Allowing any XML element to be used as the document root (the `<jsp:root>` element is no longer required)
- Support for the new `<jsp:element>` and `<jsp:attribute>` standard tag elements

The chapter includes the following sections:

- Introducing JSP Documents and XML Views
- Working with JSP Documents
- Understanding the JSP XML View

For information about additional JSP support for XML and XSL, furnished in OC4J through custom tags, refer to the *Oracle 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/
```

Introducing JSP Documents and XML Views

The term *JSP document* refers to a JSP page written in XML syntax. A JSP 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 standard action tags and custom tag libraries used. A traditional JSP page, by contrast, is typically *not* an XML document.

What advantages do JSP documents offer?

- You can create dynamic documents that can be passed directly to the OC4J JSP container in a pure XML syntax compatible with existing XML tools

- You can output content written in XML-based languages like XHTML or SVG in JSP pages
- You can generate a JSP document from a textual representation by applying an XML transformation, like XSLT
- You can use a JSP document for data-interchange between different Web applications

Many JSP syntax elements, such as standard action tags and custom tags, are already written in XML syntax. For JSP elements that are not compliant, equivalent XML elements are provided. See "Working with JSP Documents" on page 8-3 for details.

Any of the following identify a file as a JSP document to the OC4J JSP container:

- An `<is-xml>` element within the Web module's `web.xml` deployment descriptor

  The `<is-xml>` is a subelement of the `<jsp-property-group>` element used to identify files as JSP documents. For example, this `web.xml` snippet identifies files with an `.svg` extension - specified in the `<url-pattern>` subelelement - as JSP documents:

```
<web-app ...>
 <jsp-config>
  <jsp-property-group>
   <description>Define files with SVG extension as JSP documents</description>
   <url-pattern>*.svg</url-pattern>
   <is-xml>true</is-xml>
  </jsp-property-group>
 </jsp-config>
</web-app>
```

  Note that the `<is-xml>` definition overrides any of the other indicators listed below.

- A `.jspx` file extension

  Support for this extension is new in JSP 2.0, and explicity defines the file as a JSP document. Note that tag files can also be written in XML syntax, and are identified to the container by the `.tagx` extension. See "What Are Tag Files?" on page 7-19 for more on tag files.

- A `<jsp:root>` element as the top element within the document body

  This element includes a namespace specification for the JSP XML core syntax and namespace specifications for any custom tag libraries that are used. Note that this element was mandatory in JSP 1.2, but is optional in JSP 2.0, which allows you to specify your own root element.

The semantic model for JSP documents is the same as for traditional pages. A JSP 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 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.

The *XML view* is an XML document derived from a JSP page that is used to validate the page. The OC4J JSP container generates the XML view during JSP translation. The

JSP 2.0 specification defines the view as "the mapping between a JSP page, written in either XML syntax or traditional syntax, and an XML document describing it".

Beginning with the JSP1.2 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 the case of a JSP 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 difference is that ID attributes for improved error reporting are added to all XML elements.

In the case of a traditional JSP page, the Web container performs a series of transformations to create the XML view from the page. See "Understanding the JSP XML View" on page 8-10 for details.

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 Web container, for use in page validation.

Working with JSP Documents

This section describes the syntax of JSP 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 document.

JSP XML syntax includes the following elements. Note that the syntax includes an XML equivalent or replacement element for every JSP element except for `<%-- comment --%>`.

- An optional `<jsp:root>` element, which includes a namespace specification for the JSP XML core syntax and namespace specifications for any custom tag libraries that are used. Note that this element is optional in JSP 2.0, which allows you to specify your own root element.
- JSP `page` and `include` directives
- JSP declaration elements
- JSP expression elements
- JSP scriptlet elements
- JSP standard action elements
- JSP custom action elements, including JavaServer Pages Standard Tag Library (JSTL) tag elements
- New `<jsp:attribute>` and `<jsp:element>` elements, which may include expression language (EL) expressions in element bodies and attribute values
- A text element, `<jsp:text>`, for static template data

- Other XML elements, if desired, pertaining to template data

As noted previously, most of the standard JSP syntax is already XML-compliant, and can be used in a JSP document in XML element format. For the JSP elements that are not compliant, a set of equivalent XML elements are provided, as summarized in Table 8–1below.

Table 8–1 Standard JSP Syntax Versus XML Syntax

Syntax Element	JSP Syntax	XML Syntax
Comments	`<%--.. --%>`	`<!-- .. -->`
Declarations	`<%! ..%>`	`<jsp:declaration> .. </jsp:declaration>`
Include Directives	`<%@ include .. %>`	`<jsp:directive.include .. />`
Page Directives	`<%@ page .. %>`	`<jsp:directive.page .. />`
Tag Library Directives	`<%@ taglib .. %>`	`xmlns:prefix="tag library URL"`
Expressions	`<%= .. %>`	`<jsp:expression> .. </jsp:expression>`
Scriplets	`<% .. %>`	`<jsp:scriplet> .. </jsp:scriplet>`

Specifying a Document Root Element

A JSP document must have a root element. Prior to the release of JSP 2.0, the `<jsp:root>` element had to be used as the root. While this element can still be used, it is no longer required, meaning that you can specify your own root element. This flexibility allows you to take any XML document and use it as a JSP document.

The root element in the example shown below document is `html`. You can also declare tag libraries to be used in the document within the root, as discussed in the next section.

Note that neither a JSP document nor its request output is required to include an XML declaration. In fact, if the JSP document is not producing XML output, the document should not include an XML declaration.

Declaring Tag Libraries with XML Namespaces

In JSP documents, XML *namespaces* are used to include both standard actions and custom tags in the document. An XML namespace is a collection of XML element types and attribute names.

Unlike in JSP pages, standard action tags such as <jsp:useBean> are not implicitly available; instead, you must include these elements with the following namespace, which identifies the namespace of the core JSP XML syntax:

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

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 or tag file library 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 tag library. The following example declares namespaces for the JSTL prefix `c` and the custom tag prefix `my` in the `html` root element:

```
<html     xmlns:jsp="http://java.sun.com/JSP/Page"
          xmlns:c="http://java.sun.com/jsp/jstl/core"
          xmlns:my="urn:jsptagdir:/WEB-INF/tlds/mylib"
```

```
>

...body of document...

</html>
```

A URN indicates an application-relative path and must be of one of the following forms:

- "`urn:jsptld:`*`path`*", where the path points to a single tag library, in the same way as the `uri` attribute in a `taglib` directive.
- "`urn:jsptagdir:`path", where the path must start with `/WEB-INF/tags/` and points to tag extensions implemented as tag files installed in the `WEB-INF/tags/` directory or one of its subdirectories

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 TLDs in JAR files in `WEB-INF/lib` or TLDs under `WEB-INF`. See "Key web.xml Deployment Descriptor Entries" on page 7-25 and "Packaging Multiple Tag Libraries and TLD Files in a JAR File" on page 7-24.

Declaring tag libraries as `xmlns` attributes of the root element, as shown in the above example, gives all elements within the document access to those libraries. However, this is not required; tag libraries can also be referenced at the point of usage within the document, allowing you to scope a tag library for use with a particular element.

For example, the following example restricts usage of the tag library referenced by the `c` prefix to the `<c:forEach` element:

```
<c:forEach xmlns:c="http://java.sun.com/jsp/jstl/core"
var="counter" begin="0" end=${4}>
...
</c:forEach>
```

Using JSP XML Directive Elements

There are JSP XML elements that are equivalent to `page` and `include` directives. 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 language="java"
    import="com.tks.ourpackage" %>
```

This is equivalent to the following JSP XML element:

```
<jsp:directive.page language="java"
    import="com.tks.ourpackage" />
```

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.

Using 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<0) return("..."); } %>
```

This is equivalent to the following JSP XML element:

```
<jsp:declaration>
  <![CDATA[ public String func(int myint) { if (myint<0) return("..."); } ]]>
</jsp:declaration>
```

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

```
<jsp:declaration>
   public String func(int myint) { if (myint &lt; 0) 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>
   ...
```

Using 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 "Including Template and Dynamic Template Content", which follows.

Including Template and Dynamic Template Content

You can include static template text in a JSP document using either uninterpreted XML tags, which do not preserve whitespace, or with the `<jsp:text>` element. denotes template data in a JSP document:

When the OC4J 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 the use of arbitrary elements (neither standard action elements nor custom action elements) for template data wherever a `<jsp:text>` element can appear. These arbitrary elements are processed in the same way as `<jsp:text>` elements, with content being sent to the current JSP `out` object.

Consider the following JSP document source text:

```
<hello><jsp:declaration>String n="Alfred";</jsp:declaration>
<morning>
<jsp:text> Good Morning
</jsp:text>${n}
</morning>
</hello>
```

This source text results in the following output from the OC4J JSP container. Note how the whitespace is preserved:

```
<hello> <morning> Good Morning
Alfred </morning></hello>
```

You can also use `<jsp:text>` to output template data that is not well-formed XML. For example, if the `${list}` EL expression below was not wrapped in a `<jsp:text>` tag, it would be illegal in a JSP document:

```
<c:forEach var="list" begin="1" end="${4}">
  <jsp:text>${list}</jsp:text>
</c:forEach>
```

Dynamic template content can also be generated by a JSP document using EL expression, Java scripting elements (declarations, scriptlets, or expressions), standard action tags and custom tags - just as with a standard JSP page.

You can generate tags dynamically rather than hard coding them in your JSP document using EL expressions with the `<jsp:element>` standard action. This action takes one `String` attribute, which is used as the name of the generated element.

The tag element can optionally include a body, which can consist of:

- A body that does not define attributes
- One or more `<jsp:attribute>` elements, which are set as attributes of the new element
- If `<jsp:attribute>` is used, a tag body can be included within a a `<jsp:body>` element

For example, the following code in a JSP document:

```
<jsp:element name="${searchRequest.type}">
   <jsp:attribute name="language">${searchRequest.language}</jsp:attribute>
   <jsp:body>${searchRequest.content}</jsp:body>
</jsp:element>
```

Could generate the following:

```
<standardSearch language="English">What is an attribute?</standardSearch>
```

Sample Comparison: Traditional JSP Page Versus JSP XML Document

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

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

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

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

Sample Traditional JSP Page

Here is the sample page in traditional syntax:

```
<%@ taglib prefix="tags" tagdir="/WEB-INF/tags" %>
<html>
<head>
<title>An eStore for all occasions</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
</head>
<body>
<table align="center" width="100%" height="100%" border="4" bgcolor="#FFFFFF"
 bordercolor="#FF0000">
 <tr>
  <td width="64%" valign="middle" align="center" bgcolor="#FFCCCC">
   <div align="center"><img src="gifs/christmas/Electronic.gif"></div>
   <tags:ProductDetails occasion="Christmas" category="Cards" thBgColor="#FF3366"
    thFontColor="#FFFFFF">
    <jsp:attribute name="normalPrice">
    <td width="50%" align="center"><font
     color="#CC3300"><b>${name}</b></font></td>
    <td width="30%" align="center"><font
     color="#CC3300"><b>${price}</b></font></td>
    <td width="20%" align="center"><img src=${image} width="100" height="40"></td>
  </tr>
  </jsp:attribute>
  <jsp:attribute name="onSale">
 <tr>
```

```
    <td width="50%" align="center"><font color="#CC3300"><b>${name}<img
     src="gifs/main/discount.gif"></b></font></td>
     <td width="30%" align="center"><font color="red"><b>
      <strike>Was:  ${price}</strike></b></font><br><font color="#996600"><b>
      <font color="#CC3300">Now: ${saleprice}</font></b></font>
     </td>
     <td width="20%" align="center"><img src=${image} width="100"
      height="40"></td>
    </tr>
    </jsp:attribute>
   </tags:ProductDetails>
 <br><br>
  </td>
 </tr>
</table>
</body>
</html>
```

Sample JSP Document

Here is the same page in XML syntax. Note how HTML to be passed to the browser is enclosed in `<jsp:text>` tags.

```
<html xmlns:jsp="http://java.sun.com/JSP/Page"
      xmlns:tags="/WEB-INF/tags">
<jsp:directive.page contentType="text/html" />
<jsp:text>
<![CDATA[
<head><title>An eStore for all occasions</title></head>
<body>
<table align="center" width="100%" height="100%" border="4" bgcolor="#FFFFFF"
  bordercolor="#FF0000">
 <tr>
  <td width="64%" valign="middle" align="center" bgcolor="#FFCCCC">
   <div align="center"><img src="gifs/christmas/Electronic.gif"></div>
   ]]>
   </jsp:text>
   <tags:ProductDetails occasion="Christmas" category="Cards" thBgColor="#FF3366"
  thFontColor="#FFFFFF">
    <jsp:attribute name="normalPrice">
    <jsp:text>
    <![CDATA[<tr>
    <td width="50%" align="center"><font
     color="#CC3300"><b>${name}</b></font></td>
    <td width="30%" align="center"><font
     color="#CC3300"><b>${price}</b></font></td>
    <td width="20%" align="center"><img src=${image} width="100" height="40"></td>
  </tr>]]>
  </jsp:text>
  </jsp:attribute>
  <jsp:attribute name="onSale">
  <jsp:text>
  <![CDATA[<tr>
    <td width="50%" align="center"><font color="#CC3300"><b>${name}<img
     src="gifs/main/discount.gif"></b></font></td>
     <td width="30%" align="center"><font color="red"><b>
      <strike>Was:  ${price}</strike></b></font><br><font color="#996600"><b>
      <font color="#CC3300">Now: ${saleprice}</font></b></font>
     </td>
     <td width="20%" align="center"><img src=${image} width="100"
      height="40"></td>
```

```
    </tr>]]>
    </jsp:text>
    </jsp:attribute>
   </tags:ProductDetails>
  <jsp:text>
  <![CDATA[<br><br>
  </td>
 </tr>
</table>
</body>]]>
</jsp:text>
</html>
```

Understanding the JSP XML View

When OC4J 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 document—into an XML document that describes it.

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 Web container executes the following transformations in creating the XML view, both for traditional JSP pages and for JSP documents:

- The container expands the XML view to include files brought in through `include` directives.
- A Web 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 8-11.

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

The Web 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 "Declaring Tag Libraries with XML Namespaces" on page 8-4.
- It converts each `taglib` directive into an additional `xmlns` attribute in the `<jsp:root>` element. See "Declaring Tag Libraries with XML Namespaces" on page 8-4.
- It converts each `page` directive into the equivalent element in JSP XML syntax. See "Using JSP XML Directive Elements" on page 8-5.

- It converts each declaration, expression, and scriptlet into the equivalent element in JSP XML syntax. See "Using JSP XML Declaration, Expression, and Scriptlet Elements" on page 8-6.
- It converts request-time expressions into XML syntax. See "Using JSP XML Standard Action and Custom Action Elements" on page 8-6.
- It creates `<jsp:text>` elements for template data. See "Including Template and Dynamic Template Content" on page 8-7.
- 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 "Including Template and Dynamic Template Content" on page 8-7, 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 a `jsp:id` attribute that the Web container can add to each XML element in the XML view. The `jsp:id` attributes 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 Web 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. 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:

```
<%@ page import="java.util.*" %>
<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=""><![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="0"><![CDATA[
            }
            ]]></jsp:scriptlet>
   <jsp:text jsp:id=""><![CDATA[
       </BODY>
  </HTML>

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

9

JSP Globalization Support in Oracle

The Web 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 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-) for traditional JSP pages; it is `UTF-8` or `UTF-16` for JSP XML documents.
- The default for the response character encoding is `ISO-8859-1` for traditional JSP pages; it is `UTF-8` or `UTF-16` for JSP XML documents.

The determination of `UTF-8` versus `UTF-16` is according to "Autodetection of Character Encodings" in the XML specification, at `http://www.w3.org/TR/REC-xml.html`.

Be aware, however, that there is a relationship between `pageEncoding` and `contentType` regarding character encodings, as documented in the following table.

	contentType Encoding Is Specified	contentType Encoding Is Not Specified
pageEncoding Is Specified	Page source encoding is according to `pageEncoding`. Response encoding is according to `contentType`.	Page source encoding is according to `pageEncoding`. Response encoding is according to `pageEncoding`.
pageEncoding Is Not Specified	Page source encoding is according to `contentType`. Response encoding is according to `contentType`.	Page source encoding is according to the default. Response encoding is according to the default.

Be aware of the following important usage notes.

- A `page` directive that sets `contentType` or `pageEncoding` should appear as early as possible in the JSP page.
- When a page is a JSP XML document, any `pageEncoding` setting is ignored. The Web 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="2.0">
<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-` 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-` 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 Web 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 9-4.

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.

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.

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

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()
```

A

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

Apache

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

```
Copyright (c) 2000-2004 The Apache Software Foundation.
```

License

```
                              Apache License
                        Version 2.0, January 2004
                     http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition
   "control" means (i) the power, direct or indirect, to cause the
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
```

exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work

or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

   You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or

```
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

   To apply the Apache License to your work, attach the following
   boilerplate notice, with the fields enclosed by brackets "[]"
   replaced with your own identifying information. (Don't include
   the brackets!) The text should be enclosed in the appropriate
   comment syntax for the file format. We also recommend that a
   file or class name and description of purpose be included on the
   same "printed page" as the copyright notice for easier
   identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

Notice

```
This product includes software developed by
The Apache Software Foundation (http://www.apache.org/).

Portions of this software were developed at the National Center
for Supercomputing Applications (NCSA) at the University of
Illinois at Urbana-Champaign.

This software contains code derived from the RSA Data Security
Inc. MD5 Message-Digest Algorithm, including various
modifications by Spyglass Inc., Carnegie Mellon University, and
Bell Communications Research, Inc (Bellcore).

Regular expression support is provided by the PCRE library package,
which is open source software, written by Philip Hazel, and copyright
by the University of Cambridge, England. The original software is
available from
   ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/
```

Index

Symbols

<init-param>, 3-8

A

action tags
forward tag, 1-15
getProperty tag, 1-14
in JSP XML pages, 8-6
include tag, 1-14
overview of standard actions, 1-11
param tag, 1-14
plugin tag, 1-16
setProperty tag, 1-13
useBean tag, 1-12
activation.jar, Java activation files for e-mail, 3-10
addclasspath, ojspc option, 4-5
application object (implicit), 1-10
application root functionality, 6-1
application scope (JSP objects), 1-11
Application Server Control Console
using to configure the JSP container, 3-7
application-relative path, 1-23
appRoot, ojspc option, 4-5
archives
batch pretranslation of, 4-1

B

batch pretranslation
ojspc -batchMask option, 4-6
ojspc -deleteSource option, 4-7
overview of ojspc batch features, 4-2
batchMask, ojspc option, 4-6
binary data, reasons to avoid in JSP, 6-13

C

cache.jar, for Java Object Cache, 3-10
caching support, overview, 2-6
caching TLDs, 3-6
call servlet from JSP, JSP from servlet, 6-16
checker pages, 6-7
class naming, translator, 5-3
classesXX.zip, for JDBC, 3-10
classpath
JSP classpath functionality, 6-2
code, generated by translator, 5-1
comments (in JSP code), 1-8
compilation
default settings, related options, 3-9
in-process vs. out-of-process, 3-9
config object (implicit), 1-10
configuration
key JAR and ZIP files, 3-10
setting JSP configuration parameters, 3-8
content type settings
dynamic (setContentType method), 9-4
static (page directive), 9-1
context path, 6-1
context-relative path, 1-23
custom tags--see tag libraries

D

debugging
through JDeveloper, 2-7
declarations
member variables, 1-7
method variable vs. member variable, 6-9
XML declaration elements, 8-6
deleteSource, ojspc option, 4-7
demo location, OTN, 1-1
directives
include directive, 1-6
overview, 1-5
page directive, 1-5
taglib directive, 1-6
XML directive elements, 8-5
DMS support, 2-5
dynamic include
action tag, 1-14
for large static content, 6-8
logistics, 6-6
vs. static include, 6-6
Dynamic Monitoring Service--see DMS

E

EJB
interact with JSPs, 1-2
error processing (runtime), 6-19

event-handling
  with HttpSessionBindingListener, 2-8
exception object (implicit), 1-10
execution models for JSP pages, 1-21
execution of a JSP page, 1-21
explicit JSP objects, 1-9
expression language
  disabling in JSPs, 1-21
  functions, 1-20
  overview, 1-18
  syntax, 1-18
expressions
  expression syntax, 1-7
  XML expression elements, 8-6
extensions
  DMS support, 2-5
  overview of caching support, 2-6
  overview of global includes, 2-5
  overview of Oracle-specific extensions, 2-4
  overview of programmatic extensions, 2-3
  summary of portable extensions, 2-5
external resource file
  for static text, 6-8

F

fallback tag (with plugin tag), 1-16
file naming conventions, JSP files, 6-12
files
  generated by translator, 5-4
  key JAR and ZIP files, 3-10
  locations, translator output, 5-5
forward tag, 1-15

G

generated code, by translator, 5-1
generated output names, by translator, 5-2
getProperty tag, 1-14
global includes (Oracle extension)
  general use, 5-5
globalization support
  charset settings of JSP writer, 9-4
  content type settings (dynamic), 9-4
  content type settings (static), 9-1
  multibyte parameter encoding, 9-5
  overview, 9-1

H

hiding JSP pages (e.g., MVC architecture), 6-15
HttpSessionBindingListener, 2-8

I

id attribute (XML view), 8-11
implicit JSP objects
  overview, 1-9
  using implicit objects, 1-10
imports, default packages, 6-3
include directive, 1-6
include tag, 1-14
interaction, JSP-servlet, 6-16
invoke servlet from JSP, JSP from servlet, 6-16

J

J2EE
  definition, 2-1
JavaBeans
  use with useBean tag, 1-12
  vs. scriptlets, 6-5
JDeveloper
  JSP support, 2-7
JDK, 2-1
JDK 1.4 considerations, 6-4
jndi.jar, for data sources and EJBs, 3-10
JSP container
  configuration parameters, 3-1
  configuring through Application Server Control Console, 3-7
  configuring through XML files, 3-8
  setting initialization paramaters, 3-8
jsp fallback tag (with plugin tag), 1-16
jsp forward tag, 1-15
jsp getProperty tag, 1-14
jsp id attribute (XML view), 8-11
jsp include tag, 1-14
JSP pages
  interact with EJBs, 1-2
  overview, 1-1
  pretranslating using ojspc, 4-1
  simple example code, 1-2
jsp param tag, 1-14
jsp plugin tag, 1-16
jsp setProperty tag, 1-13
JSP technology
  overview, 1-1
jsp text element (XML syntax), 8-7
jsp useBean tag
  syntax, 1-12
JSP XML document, 8-1
JSP XML syntax--see XML syntax
JSP XML view--see XML view
jsp-cache-directory setting, 5-5
jsp-cache-tlds flag, 7-27
JSPs
  adding to a deployed application, 3-9
  modifying in a deployed application, 3-9
JSP-servlet interaction
  invoking JSP from servlet, request dispatcher, 6-17
  invoking servlet from JSP, 6-17
  passing data, JSP to servlet, 6-17
  passing data, servlet to JSP, 6-18
  sample code, 6-19
JspWriter object, 1-10
JSTL, overview of support, 2-6
jta.jar, for Java Transaction API, 3-10
justrun mode, 3-3
JVM, 2-1

M

mail.jar, for e-mail from applications, 3-10
member variable declarations, 6-9
method variable declarations, 6-9
Model-View-Controller, hiding JSP pages, 6-15
multibyte parameter encoding
  general/standard, 9-5
MVC architecture, hiding JSP pages, 6-15

N

naming conventions, JSP files, 6-12
National Language Support--see Globalization Support
NLS--see Globalization Support

O

objects and scopes (JSP objects), 1-8
ojspc pretranslation tool
  command-line syntax, 4-2
  option summary table, 4-4
  overview, 4-1
  overview of basic functionality, 4-1
  overview of batch pretranslation, 4-2
ojsp.jar, for JSP container, 3-10
ojsputil.jar, for JSP tag libraries and utilities, 3-10
on-demand translation (runtime), 1-22
Oracle platforms supporting JSP
  JDeveloper, 2-7
out object (implicit), 1-10
output files
  generated by translator, 5-4
  locations, 5-5
output names, conventions, 5-2

P

package imports, default, 6-3
package naming
  by translator, 5-3
page directive
  characteristics, 6-10
  contentType setting for globalization support, 9-1
  overview, 1-5
page implementation class
  generated code, 5-1
  overview, 1-22
page object (implicit), 1-9
page scope (JSP objects), 1-11
pageContext object (implicit), 1-9
page-relative path, 1-23
param tag, 1-14
performance
  use of pretranslation, 6-14
persistent caching for TLD files, 7-26
plugin tag, 1-16
pretranslation
  ojspc utility, 4-1
programming considerations
  general strategies, 6-4

R

recompile mode, 3-3
reducing generated tag code, 3-4
reloading JSP-generated classes, 3-3
request dispatcher (JSP-servlet interaction), 6-17
request objects
  JSP implicit request object, 1-9
request scope (JSP objects), 1-11
RequestDispatcher interface, 6-17
requesting a JSP page, 1-23
resource management
  overview of JSP extensions, 2-11
  standard session management, 2-8
response objects
  JSP implicit response object, 1-9
retranslating JSP pages, 3-3
retranslation or reloading at runtime, 3-9

S

sample applications
  demo location, OTN, 1-1
  HttpSessionBindingListener sample, 2-8
  JSP-servlet interaction, 6-19
  traditional vs. XML syntax, 8-8
  transformation to XML view, 8-11
scopes (JSP objects), 1-10
scripting elements
  comments, 1-8
  declarations, 1-7
  expressions, 1-7
  overview, 1-6
  scriptlets, 1-7
scripting variables (tag libraries)
  declaration through TEI class, 7-11
  declaration through TLD, 7-10
  scopes, 7-9
  using, 7-9
scriptlets
  scriptlet syntax, 1-7
  vs. JavaBeans, 6-5
  XML scriptlet elements, 8-6
servlet path, 6-1
servlet-JSP interaction
  invoking JSP from servlet, request dispatcher, 6-17
  invoking servlet from JSP, 6-17
  passing data, JSP to servlet, 6-17
  passing data, servlet to JSP, 6-18
  sample code, 6-19
session events
  with HttpSessionBindingListener, 2-8
session objects
  JSP implicit session object, 1-9
session scope (JSP objects), 1-11
setCharacterEncoding() method, 9-5
setContentType() method, globalization support, 9-4

setProperty tag, 1-13
setting initialization parameters, 3-8
setWriterEncoding() method, globalization support, 9-4
sharing tag libraries, 3-6
simple tag handlers, 7-7
SimpleTag interface, 7-7
static include
  directive, 1-6
  logistics, 6-6
  vs. dynamic include, 6-6
static text
  external resource file, 6-8
  in member variables, 5-2
  workaround for large static content, 6-8
syntax (overview), 1-4

T

tag files, 7-18
tag handler reuse, 3-5
tag handlers (tag libraries)
  access to outer tag handlers, 7-12
  accessing body content, 7-5
  body processing, 7-4
  OC4J tag handler code generation, 7-18
  OC4J tag handler instance reuse / pooling, 7-16
  overview, 7-4
tag libraries
  multiple tag libraries in a JAR file, 7-24
  overview of functionality, 1-17
  persistent caching for TLD files, 7-26
  sharing across applications, 7-26
  strategy, when to create, 7-2
  tag handlers, 7-3
  well-known location, 7-26
tag library descriptor files
  persistent caching, 7-26
  specifying TLDs for multiple tag libraries in a JAR file, 7-24
tag pooling, 7-16
tag-extra-info classes (tag libraries)
  general use, getVariableInfo() method, 7-11
taglib directive
  syntax, 1-6
template data, 8-2
text element (XML syntax), 8-7
tips
  avoid JSP use with binary data, 6-13
  JavaBeans vs. scriptlets, 6-5
  JSP preservation of white space, 6-12
  method vs. member variable declaration, 6-9
  page directive characteristics, 6-10
  static vs. dynamic includes, 6-6
  using a "checker" page, 6-7
  when to create tag libraries, 7-2
  workaround, large static content, 6-8
TLD caching, 3-6
TLD validation, 3-3
translation, on-demand (runtime), 1-22
translator
  generated class names, 5-3
  generated code features, 5-1
  generated files, 5-4
  generated member variables, static text, 5-2
  generated names, general conventions, 5-2
  generated package names, 5-3
  Oracle JSP global includes, 5-5
  output file locations, 5-5
  pretranslation using ojspc, 4-1

U

useBean tag, 1-12

V

variable element (tag libraries), 7-10

W

Web module
  adding JSPs to a, 3-9
  modifying JSPs in a, 3-9
well-known location (tag libraries), 7-26
well-known tag libraries, 3-6

X

XML support
  JSP XML document, 8-1
  JSP XML documents and JSP XML view, overview, 8-1
  JSP XML syntax, 8-3
  XML view, 8-10
XML syntax
  custom action elements, 8-6
  declaration elements, 8-6
  directive elements, 8-5
  expression elements, 8-6
  sample, traditional vs. XML syntax, 8-8
  scriptlet elements, 8-6
  standard action elements, 8-6
  text element and other elements, 8-7
XML view
  jsp id attribute for validation, 8-11
  sample transformation, 8-11
  transformation from JSP page to XML view, 8-10
xmlparserv2.jar, for XML validation, 3-10