JavaServer Pages™ Specification

Version 2.0

please send comments to jsp-spec-comments@eng.sun.com

Final Release - November 24, 2003

Mark Roth
Eduardo Pelegrí-Llopart

4150 Network Circle
Santa Clara, CA 95054, USA
650 960-1300 fax: 650 969-9131

JavaServer Pages™ Specification (“Specification”)
Version: 2.0
Status: FCS
Release: November 24, 2003

NOTICE; LIMITED LICENSE GRANTS

Sun Microsystems, Inc. (“Sun”) hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense), under the Sun’s applicable intellectual property rights to view, download, use and reproduce the Specification only for the purpose of internal evaluation, which shall be understood to include developing applications intended to run on an implementation of the Specification provided that such applications do not themselves implement any portion(s) of the Specification.

Sun also grants you a perpetual, non-exclusive, worldwide, fully paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights or patent rights it may have in the Specification to create and/or distribute an Independent Implementation of the Specification that: (i) fully implements the Spec(s) including all its required interfaces and functionality; (ii) does not modify, subset, superset or otherwise extend the Licensor Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the Licensor Name Space other than those required/authorized by the Specification or Specifications being implemented; and (iii) passes the TCK (including satisfying the requirements of the applicable TCK Users Guide) for such Specification. The foregoing license is expressly conditioned on your not acting outside its scope. No license is granted hereunder for any other purpose.

You need not include limitations (i)-(iii) from the previous paragraph or any other particular “pass through” requirements in any license You grant concerning the use of your Independent Implementation or products derived from it. However, except with respect to implementations of the Specification (and products derived from them) that satisfy limitations (i)-(iii) from the previous paragraph, You may neither: (a) grant or otherwise pass through to your licensees any licenses under Sun’s applicable intellectual property rights; nor (b) authorize your licensees to make any claims concerning their implementation’s compliance with the Spec in question.

For the purposes of this Agreement: “Independent Implementation” shall mean an implementation of the Specification that neither derives from any of Sun’s source code or binary code materials nor, except with an appropriate and separate license from Sun, includes any of Sun’s source code or binary code materials; and “Licensor Name Space” shall mean the public class or interface declarations whose names begin with “java”, “javax”, “com.sun” or their equivalents in any subsequent naming convention adopted by Sun through the Java Community Process, or any recognized successors or replacements thereof.

This Agreement will terminate immediately without notice from Sun if you fail to comply with any material provision of or act outside the scope of the licenses granted above.

TRADEMARKS

No right, title, or interest in or to any trademarks, service marks, or trade names of Sun or Sun’s licensors is granted hereunder. Sun, Sun Microsystems, the Sun logo, Java, the Java Coffee Cup logo, JSP, and JavaServer Pages are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED “AS IS”. SUN MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT, THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE OR THAT ANY PRACTICE OR IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADE SECRETS OR OTHER RIGHTS. This document does not represent any commitment to release or implement any portion of the Specification in any product.

THE SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION THEREIN; THESE CHANGES WILL BE INCORPORATED INTO NEW VERSIONS OF THE SPECIFICATION, IF ANY. SUN MAY MAKE IMPROVEMENTS AND/OR CHANGES TO THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THE SPECIFICATION AT ANY TIME. Any use of such

changes in the Specification will be governed by the then-current license for the applicable version of the Specification.

LIMITATION OF LIABILITY

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO ANY FURNISHING, PRACTICING, MODIFYING OR ANY USE OF THE SPECIFICATION, EVEN IF SUN AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will indemnify, hold harmless, and defend Sun and its licensors from any claims arising or resulting from: (i) your use of the Specification; (ii) the use or distribution of your Java application, applet and/or clean room implementation; and/or (iii) any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND

U.S. Government: If this Specification is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Specification and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

REPORT

You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your use of the Specification ("Feedback"). To the extent that you provide Sun with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Sun a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.

(LFI#136181/Form ID#011801)

Contents

Preface . . . . . **xix**

Status . . . . . **xxvii**

Overview . . . . . **xxix**

The JavaServer Pages™ Technology xxix

Basic Concepts xxxi

Users of JavaServer Pages xxxiii

Part I **1-1**

JSP.1 Core Syntax and Semantics **1-3**

JSP.1.1 What Is a JSP Page 1-3

JSP.1.1.1 Web Containers and Web Components 1-3

JSP.1.1.2 Generating HTML 1-4

JSP.1.1.3 Generating XML 1-4

JSP.1.1.4 Translation and Execution Phases 1-4

JSP.1.1.5 Validating JSP pages 1-5

JSP.1.1.6 Events in JSP Pages 1-6

JSP.1.1.7 JSP Configuration Information 1-6

JSP.1.1.8 Naming Conventions for JSP Files 1-6

JSP.1.1.9 Compiling JSP Pages 1-7

JSP.1.1.10 Debugging JSP Pages 1-8

JSP.1.2 Web Applications 1-8

JSP.1.2.1 Relative URL Specifications 1-9

JSP.1.3 Syntactic Elements of a JSP Page 1-10

JSP.1.3.1 Elements and Template Data 1-10

JSP.1.3.2 Element Syntax 1-10

JSP.1.3.3 Start and End Tags 1-11

JSP.1.3.4 Empty Elements 1-12

JSP.1.3.5 Attribute Values 1-12

JSP.1.3.6 The jsp:attribute, jsp:body and jsp:element Elements 1-12

JSP.1.3.7 Valid Names for Actions and Attributes 1-14

JSP.1.3.8 White Space 1-14

JSP.1.3.9 JSP Documents 1-15

JSP.1.3.10 JSP Syntax Grammar 1-16
JSP.1.4 Error Handling 1-33
JSP.1.4.1 Translation Time Processing Errors 1-33
JSP.1.4.2 Request Time Processing Errors 1-33
JSP.1.4.3 Using JSPs as Error Pages 1-34
JSP.1.5 Comments 1-34
JSP.1.5.1 Generating Comments in Output to Client . . . 1-34
JSP.1.5.2 JSP Comments 1-35
JSP.1.6 Quoting and Escape Conventions 1-35
JSP.1.7 Overall Semantics of a JSP Page 1-37
JSP.1.8 Objects 1-38
JSP.1.8.1 Objects and Variables 1-38
JSP.1.8.2 Objects and Scopes 1-39
JSP.1.8.3 Implicit Objects 1-40
JSP.1.8.4 The pageContext Object 1-42
JSP.1.9 Template Text Semantics 1-42
JSP.1.10 Directives 1-42
JSP.1.10.1 The **page** Directive 1-43
JSP.1.10.2 The **taglib** Directive 1-49
JSP.1.10.3 The **include** Directive 1-51
JSP.1.10.4 Implicit Includes 1-52
JSP.1.10.5 Including Data in JSP Pages 1-52
JSP.1.10.6 Additional Directives for Tag Files 1-54
JSP.1.11 EL Elements 1-54
JSP.1.12 Scripting Elements 1-54
JSP.1.12.1 Declarations 1-55
JSP.1.12.2 Scriptlets 1-56
JSP.1.12.3 Expressions 1-57
JSP.1.13 Actions 1-58
JSP.1.14 Tag Attribute Interpretation Semantics 1-58
JSP.1.14.1 Request Time Attribute Values 1-58
JSP.1.14.2 Type Conversions 1-59

JSP.2 Expression Language 1-63

JSP.2.1 Overview 1-63
JSP.2.2 The Expression Language in JSP 2.0 1-64
JSP.2.2.1 Expressions and Attribute Values 1-64
JSP.2.2.2 Expressions and Template Text 1-65
JSP.2.2.3 Implicit Objects 1-66
JSP.2.2.4 Deactivating EL Evaluation 1-67

JSP.2.2.5 Disabling Scripting Elements 1-67
JSP.2.3 General Syntax of the Expression Language 1-67
JSP.2.3.1 Overview 1-68
JSP.2.3.2 Literals 1-68
JSP.2.3.3 Errors, Warnings, Default Values 1-68
JSP.2.3.4 Operators "[]" and "." 1-68
JSP.2.3.5 Arithmetic Operators 1-69
JSP.2.3.6 Logical Operators 1-73
JSP.2.3.7 Empty Operator - empty A 1-73
JSP.2.3.8 Conditional Operator - A ? B : C 1-74
JSP.2.3.9 Parentheses 1-74
JSP.2.3.10 Operator Precedence 1-74
JSP.2.4 Reserved Words 1-75
JSP.2.5 Named Variables 1-75
JSP.2.6 Functions 1-75
JSP.2.6.1 Invocation Syntax 1-76
JSP.2.6.2 Tag Library Descriptor Information 1-76
JSP.2.6.3 Example 1-77
JSP.2.6.4 Semantics 1-77
JSP.2.7 Implicit Objects 1-78
JSP.2.8 Type Conversion 1-78
JSP.2.8.1 To Coerce a Value X to Type Y 1-78
JSP.2.8.2 Coerce A to String 1-79
JSP.2.8.3 Coerce A to Number type N 1-79
JSP.2.8.4 Coerce A to Character 1-80
JSP.2.8.5 Coerce A to Boolean 1-80
JSP.2.8.6 Coerce A to Any Other Type T 1-80
JSP.2.9 Collected Syntax 1-81

JSP.3 JSP Configuration 1-85

JSP.3.1 JSP Configuration Information in web.xml 1-85
JSP.3.2 Taglib Map 1-85
JSP.3.3 JSP Property Groups 1-86
JSP.3.3.1 JSP Property Groups 1-86
JSP.3.3.2 Deactivating EL Evaluation 1-87
JSP.3.3.3 Disabling Scripting Elements 1-89
JSP.3.3.4 Declaring Page Encodings 1-89
JSP.3.3.5 Defining Implicit Includes 1-90
JSP.3.3.6 Denoting XML Documents 1-91

JSP.4 Internationalization Issues 1-93
JSP.4.1 Page Character Encoding 1-94
JSP.4.2 Response Character Encoding 1-95
JSP.4.3 Request Character Encoding 1-96
JSP.4.4 XML View Character Encoding 1-96
JSP.4.5 Delivering Localized Content 1-96
JSP.5 Standard Actions 1-99
JSP.5.1 <jsp:useBean> 1-99
JSP.5.2 <jsp:setProperty> 1-105
JSP.5.3 <jsp:getProperty> 1-107
JSP.5.4 <jsp:include> 1-109
JSP.5.5 <jsp:forward> 1-110
JSP.5.6 <jsp:param> 1-112
JSP.5.7 <jsp:plugin> 1-112
JSP.5.8 <jsp:params> 1-115
JSP.5.9 <jsp:fallback> 1-115
JSP.5.10 <jsp:attribute> 1-115
JSP.5.11 <jsp:body> 1-118
JSP.5.12 <jsp:invoke> 1-119
JSP.5.12.1 Basic Usage 1-119
JSP.5.12.2 Storing Fragment Output 1-119
JSP.5.12.3 Providing a Fragment Access to Variables . . 1-120
JSP.5.13 <jsp:doBody> 1-121
JSP.5.14 <jsp:element> 1-122
JSP.5.15 <jsp:text> 1-124
JSP.5.16 <jsp:output> 1-125
JSP.5.17 Other Standard Actions 1-129
JSP.6 JSP Documents 1-131
JSP.6.1 Overview of JSP Documents and of XML Views . . . 1-131
JSP.6.2 JSP Documents 1-133
JSP.6.2.1 Identifying JSP Documents 1-133
JSP.6.2.2 Overview of Syntax of JSP Documents 1-134
JSP.6.2.3 Semantic Model 1-135
JSP.6.2.4 JSP Document Validation 1-136
JSP.6.3 Syntactic Elements in JSP Documents 1-136
JSP.6.3.1 Namespaces, Standard Actions, and Tag Libraries 1-136
JSP.6.3.2 The jsp:root Element 1-137

JSP.6.3.3 The jsp:output Element 1-138
JSP.6.3.4 The jsp:directive.page Element 1-139
JSP.6.3.5 The jsp:directive.include Element 1-139
JSP.6.3.6 Additional Directive Elements in Tag Files . 1-139
JSP.6.3.7 Scripting Elements 1-139
JSP.6.3.8 Other Standard Actions 1-140
JSP.6.3.9 Template Content 1-140
JSP.6.3.10 Dynamic Template Content 1-141
JSP.6.4 Examples of JSP Documents 1-142
JSP.6.4.1 Example: A simple JSP document 1-142
JSP.6.4.2 Example: Generating Namespace-aware documents 1-143
JSP.6.4.3 Example: Generating non-XML documents 1-143
JSP.6.4.4 Example: Using Custom Actions and Tag Files 1-145
JSP.6.5 Possible Future Directions for JSP documents 1-146
JSP.6.5.1 Generating XML Content Natively 1-146
JSP.6.5.2 Schema and XInclude Support 1-147

JSP.7 Tag Extensions 1-149

JSP.7.1 Introduction 1-149
JSP.7.1.1 Goals 1-150
JSP.7.1.2 Overview 1-151
JSP.7.1.3 Classic Tag Handlers 1-152
JSP.7.1.4 Simple Examples of Classic Tag Handlers . 1-152
JSP.7.1.5 Simple Tag Handlers 1-154
JSP.7.1.6 JSP Fragments 1-156
JSP.7.1.7 Simple Examples of Simple Tag Handlers . 1-156
JSP.7.1.8 Attributes With Dynamic Names 1-158
JSP.7.1.9 Event Listeners 1-158
JSP.7.2 Tag Libraries 1-158
JSP.7.2.1 Packaged Tag Libraries 1-158
JSP.7.2.2 Location of Java Classes 1-159
JSP.7.2.3 Tag Library directive 1-159
JSP.7.3 The Tag Library Descriptor 1-160
JSP.7.3.1 Identifying Tag Library Descriptors 1-160
JSP.7.3.2 TLD resource path 1-161
JSP.7.3.3 Taglib Map in web.xml 1-161
JSP.7.3.4 Implicit Map Entries from TLDs 1-162
JSP.7.3.5 Implicit Map Entries from the Container . . . 1-162

JSP.7.3.6 Determining the TLD Resource Path 1-162
JSP.7.3.7 Translation-Time Class Loader 1-164
JSP.7.3.8 Assembling a Web Application 1-164
JSP.7.3.9 Well-Known URIs 1-165
JSP.7.3.10 Tag and Tag Library Extension Elements . . 1-165
JSP.7.4 Validation 1-169
JSP.7.4.1 Translation-Time Mechanisms 1-169
JSP.7.4.2 Request-Time Errors 1-170
JSP.7.5 Conventions and Other Issues 1-171
JSP.7.5.1 How to Define New Implicit Objects 1-171
JSP.7.5.2 Access to Vendor-Specific information 1-172
JSP.7.5.3 Customizing a Tag Library 1-172

JSP.8 Tag Files 1-173

JSP.8.1 . . . . . Overview 1-173
JSP.8.2 Syntax of Tag Files 1-174
JSP.8.3 Semantics of Tag Files 1-174
JSP.8.4 Packaging Tag Files 1-176
JSP.8.4.1 Location of Tag Files 1-176
JSP.8.4.2 Packaging in a JAR 1-176
JSP.8.4.3 Packaging Directly in a Web Application . . 1-177
JSP.8.4.4 Packaging as Precompiled Tag Handlers . . . 1-178
JSP.8.5 Tag File Directives 1-179
JSP.8.5.1 The tag Directive 1-179
JSP.8.5.2 The attribute Directive 1-182
JSP.8.5.3 The variable Directive 1-183
JSP.8.6 Tag Files in XML Syntax 1-186
JSP.8.7 XML View of a Tag File 1-186
JSP.8.8 Implicit Objects 1-186
JSP.8.9 Variable Synchronization 1-188
JSP.8.9.1 Synchronization Points 1-189
JSP.8.9.2 Synchronization Examples 1-190

JSP.9 Scripting 1-195

JSP.9.1 Overall Structure 1-195
JSP.9.1.1 Valid JSP Page 1-195
JSP.9.1.2 Reserved Names 1-196
JSP.9.1.3 Implementation Flexibility 1-196
JSP.9.2 Declarations Section 1-197
JSP.9.3 Initialization Section 1-197

JSP.9.4 Main Section 1-197
JSP.9.4.1 Template Data 1-197
JSP.9.4.2 Scriptlets 1-198
JSP.9.4.3 Expressions 1-198
JSP.9.4.4 Actions 1-198

JSP.10 XML View 1-201

JSP.10.1 XML View of a JSP Document, JSP Page or Tag File . . 1-201
JSP.10.1.1 JSP Documents and Tag Files in XML Syntax . 1-201
JSP.10.1.2 JSP Pages or Tag Files in JSP Syntax 1-202
JSP.10.1.3 JSP Comments 1-203
JSP.10.1.4 The page Directive 1-203
JSP.10.1.5 The taglib Directive 1-203
JSP.10.1.6 The include Directive 1-204
JSP.10.1.7 Declarations 1-204
JSP.10.1.8 Scriptlets 1-204
JSP.10.1.9 Expressions 1-205
JSP.10.1.10 Standard and Custom Actions 1-205
JSP.10.1.11 Request-Time Attribute Expressions 1-205
JSP.10.1.12 Template Text and XML Elements 1-206
JSP.10.1.13 The jsp:id Attribute 1-207
JSP.10.1.14 The tag Directive 1-207
JSP.10.1.15 The attribute Directive 1-207
JSP.10.1.16 The variable Directive 1-207
JSP.10.2 Validating an XML View of a JSP page 1-208
JSP.10.3 Examples 1-208
JSP.10.3.1 A JSP document 1-208
JSP.10.3.2 A JSP page and its corresponding XML View . 1-209
JSP.10.3.3 Clearing Out Default Namespace on Include 1-210
JSP.10.3.4 Taglib Direcive Adds to Global Namespace 1-211
JSP.10.3.5 Collective Application of Inclusion Semantics . 1-211

Part II. 2-1

JSP.11 JSP Container 2-3

JSP.11.1 JSP Page Model 2-3

JSP.11.1.1 Protocol Seen by the Web Server 2-3
JSP.11.2 JSP Page Implementation Class 2-5
JSP.11.2.1 API Contracts 2-6
JSP.11.2.2 Request and Response Parameters 2-7
JSP.11.2.3 Omitting the extends Attribute 2-8
JSP.11.2.4 Using the extends Attribute 2-10
JSP.11.3 Buffering 2-11
JSP.11.4 Precompilation 2-12
JSP.11.4.1 Request Parameter Names 2-12
JSP.11.4.2 Precompilation Protocol 2-13
JSP.11.5 Debugging Requirements 2-13
JSP.11.5.1 Line Number Mapping Guidelines 2-14

JSP.12 Core API 2-17

JSP.12.1 JSP Page Implementation Object Contract 2-17
JSP.12.1.1 JspPage 2-17
JSP.12.1.2 HttpJspPage 2-19
JSP.12.1.3 JspFactory 2-20
JSP.12.1.4 JspEngineInfo 2-22
JSP.12.2 Implicit Objects 2-22
JSP.12.2.1 JspContext 2-22
JSP.12.2.2 PageContext 2-27
JSP.12.2.3 JspWriter 2-34
JSP.12.2.4 ErrorData 2-42
JSP.12.3 An Implementation Example 2-43
JSP.12.4 Exceptions 2-44
JSP.12.4.1 JspException 2-44
JSP.12.4.2 JspTagException 2-46
JSP.12.4.3 SkipPageException 2-47

JSP.13 Tag Extension API 2-49

JSP.13.1 Classic Tag Handlers 2-50
JSP.13.1.1 JspTag 2-53
JSP.13.1.2 Tag 2-53
JSP.13.1.3 IterationTag 2-58
JSP.13.1.4 TryCatchFinally 2-61
JSP.13.1.5 TagSupport 2-62
JSP.13.2 Tag Handlers that want Access to their Body Content 2-66
JSP.13.2.1 BodyContent 2-66
JSP.13.2.2 BodyTag 2-68

JSP.13.2.3 BodyTagSupport 2-72
JSP.13.3 Dynamic Attributes 2-74
JSP.13.3.1 DynamicAttributes 2-75
JSP.13.4 Annotated Tag Handler Management Example 2-76
JSP.13.5 Cooperating Actions 2-79
JSP.13.6 Simple Tag Handlers 2-80
JSP.13.6.1 SimpleTag 2-82
JSP.13.6.2 SimpleTagSupport 2-84
JSP.13.6.3 TagAdapter 2-86
JSP.13.7 JSP Fragments 2-88
JSP.13.7.1 JspFragment 2-91
JSP.13.8 Example Simple Tag Handler Scenario 2-92
JSP.13.9 Translation-time Classes 2-98
JSP.13.9.1 TagLibraryInfo 2-101
JSP.13.9.2 TagInfo 2-104
JSP.13.9.3 TagFileInfo 2-108
JSP.13.9.4 TagAttributeInfo 2-110
JSP.13.9.5 PageData 2-112
JSP.13.9.6 TagLibraryValidator 2-112
JSP.13.9.7 ValidationMessage 2-114
JSP.13.9.8 TagExtraInfo 2-115
JSP.13.9.9 TagData 2-117
JSP.13.9.10 VariableInfo 2-119
JSP.13.9.11 TagVariableInfo 2-122
JSP.13.9.12 FunctionInfo 2-124

JSP.14 Expression Language API 2-127

JSP.14.1 Expression Evaluator 2-127
JSP.14.1.1 ExpressionEvaluator 2-128
JSP.14.1.2 Expression 2-130
JSP.14.1.3 VariableResolver 2-130
JSP.14.1.4 FunctionMapper 2-131
JSP.14.2 Exceptions 2-132
JSP.14.2.1 ELException 2-132
JSP.14.2.2 ELParseException 2-133
JSP.14.3 Code Fragment 2-134

Part III 3-1

JSP.A Packaging JSP Pages 3-3
JSP.A.1 A Very Simple JSP Page 3-3
JSP.A.2 The JSP Page Packaged as Source in a WAR File 3-3
JSP.A.3 The Servlet for the Compiled JSP Page 3-4
JSP.A.4 The Web Application Descriptor 3-5
JSP.A.5 The WAR for the Compiled JSP Page 3-6
JSP.B JSP Elements of web.xml 3-7
JSP.B.1 XML Schema for JSP 2.0 Deployment Descriptor 3-7
JSP.C Tag Library Descriptor Formats 3-15
JSP.C.1 XML Schema for TLD, JSP 2.0 3-15
JSP.C.2 DTD for TLD, JSP 1.2 3-41
JSP.C.3 DTD for TLD, JSP 1.1 3-50
JSP.D Page Encoding Detection 3-57
JSP.D.1 Detection Algorithm 3-57
JSP.E Changes 3-61
JSP.E.1 Changes between JSP 2.0 PFD3 and JSP 2.0 Final . . . 3-61
JSP.E.2 Changes between JSP 2.0 PFD2 and JSP 2.0 PFD3 . . 3-62
JSP.E.3 Changes between JSP 2.0 PFD and JSP 2.0 PFD2 . . . 3-64
JSP.E.4 Changes between JSP 2.0 PFD1a and JSP 2.0 PFD . . 3-68
JSP.E.5 Changes between JSP 2.0 PD2 and JSP 2.0 PFD1a . . 3-70
JSP.E.6 Changes between JSP 2.0 PD1 and JSP 2.0 PD2 3-71
JSP.E.7 Changes between JSP 2.0 CD2 and JSP 2.0 PD1 3-72
JSP.E.8 Changes between JSP 2.0 CD1 and JSP 2.0 CD2 3-73
E.8.1 Between CD2c and CD2 3-73
E.8.2 Between CD2b and CD2c 3-74
E.8.3 Between CD2a and CD2b 3-74
E.8.4 Changes between CD1 and CD2a 3-75
JSP.E.9 Changes between JSP 2.0 ED1 and JSP 2.0 CD1 3-75
E.9.5 JSP Fragments, .tag Files, and Simple Tag Handlers 3-75
E.9.6 Expression Language Added 3-75
E.9.7 EBNF Fixes 3-76
E.9.8 I18N Clarifications 3-76
E.9.9 Other Changes 3-76
JSP.E.10 Changes Between JSP 1.2 Final Draft and JSP 2.0 ED1 . 3-

76
E.10.10 Typographical Fixes and Version Numbers . . 3-76
E.10.11 Added EBNF Grammar for JSP Standard Syntax 3-76
E.10.12 Added Users of JavaServer Pages Section . . . 3-77
E.10.13 Added Placeholders for Expression Language and Custom Actions Using JSP 3-77
E.10.14 Added Requirement for Debugging Support . 3-77
JSP.E.11 Changes Between PFD 2 and Final Draft 3-77
E.11.15 Added jsp:id mechanism 3-77
E.11.16 Other Small Changes 3-77
E.11.17 Clarification of role of id 3-78
E.11.18 Clarifications on Multiple Requests and Threading 3-78
E.11.19 Clarifications on JSP Documents 3-78
E.11.20 Clarifications on Well Known Tag Libraries . 3-78
E.11.21 Clarified Impact of Blocks 3-79
E.11.22 Other Small Clarifications 3-79
JSP.E.12 Changes Between 1.2 PFD 1b and PFD 2 3-80
E.12.23 Added elements to Tag Library Descriptor . . 3-80
E.12.24 Changed the way version information is encoded into TLD 3-80
E.12.25 Assigning String literals to Object attributes . 3-80
E.12.26 Clarification on valid names for prefix, action and attributes 3-81
E.12.27 Clarification of details of empty actions 3-81
E.12.28 Corrections related to XML syntax 3-81
E.12.29 Other changes 3-81
JSP.E.13 Changes Between 1.2 PFD and 1.2 PFD 1b 3-82
JSP.E.14 Changes Between 1.2 PD1 and 1.2 PFD 3-82
E.14.30 Deletions 3-83
E.14.31 Additions 3-83
E.14.32 Clarifications 3-83
E.14.33 Changes 3-84
JSP.E.15 Changes Between 1.1 and 1.2 PD1 3-84
E.15.34 Organizational Changes 3-84
E.15.35 New Document 3-85
E.15.36 Additions to API 3-85

E.15.37 Clarifications 3-86
E.15.38 Changes 3-86
JSP.E.16 Changes Between 1.0 and 1.1 3-86
E.16.39 Additions 3-86
E.16.40 Changes 3-87
JSP.F Glossary 3-89

Preface

This document is the JavaServer™ Pages 2.0 Specification (JSP 2.0).

This specification was developed following the Java Community Process (SM) (JCP). Comments from Experts, Participants, and the Public were reviewed, and improvements were incorporated into the specification where applicable.

The original Java Specification Request (JSR-152) listed the version number of the specifcation as 1.3. The scope and content of the specification effort did not change, but the expert group realized that the new features would have a deep impact in the development model of JSP applications and decided that 2.0 would more appropriately reflect that impact.

Relation To JSP 1.2

JSP 2.0 extends the JavaServer Pages 1.2 Specification (JSP 1.2) in the following ways:

- The JSP 2.0 specification requires the Java™ 2 Platform, Standard Edition version 1.3 or later for standalone containers, and version 1.4 for containers that are part of a Java 2 Enterprise Edition 1.4 environment. All JSP containers must be able to run in a J2SE 1.4 environment.
- The JSP 2.0 specification uses the Servlet 2.4 specification for its web semantics.
- A simple Expression Language (EL) has been added. The EL can be used to easily access data from the JSP pages. The EL simplifies writing *script-less* JSP pages that do not use Java scriptlets or Java expressions and thus have a more controlled interaction with the rest of the Web Application.

- New syntax elements for defining custom actions using the JSP technology directly have been added. These elements are delivered into .tag and .tagx files which can be authored by developers and page authors alike to provide encapsulation and reusability of common actions.
- The XML syntax has been clarified and improved substantially. New standard extensions have been added for JSP pages (.jspx) and for tag files (.tagx). We expect that the new mechanisms will compel authors to use the XML syntax to generate XML documents in JSP 2.0.
- An API for invoking the EL has been added. This API will likely be used in the implementation of the EL in JSP 2.0 and JSTL but we expect it to also be used in other technologies like JavaServer™ Faces.
- A new Simple Invocation Protocol has been added. This API exploits what we expect to be the prevalent use of script-less pages. The *simple* invocation protocol avoids the complex "inverted closure" mechanism of the *classic* invocation protocol introduced in JSP 1.1 and is used for implementing tag files.

Major Version Number Upgrade (JSP 2.0)

The new features introduced in this specification such as a built-in expression language, a new invocation protocol, and JSP fragments, together with the JSP Standard Tag Library, will have a substantial impact on the methodology page authors will use to write JSP pages. The impact is strong enough that the expert group felt it was appropriate to upgrade the major version number of the JSP specification to JSP 2.0.

Among other benefits, we believe this version number upgrade will help draw developer's attention to these new features. It will also allow one to more easily differentiate between the two different programming models (JSP 1.x style vs. JSP 2.x style).

Backwards Compatibility with JSP 1.2

Where possible, JSP 2.0 attempts to be fully backwards compatible with JSP 1.2. In some cases, there are ambiguities in the JSP 1.2 specification that have been clarified in JSP 2.0. Because some JSP 1.2 containers behave differently, some applications that rely on container-specific behavior may need to be adjusted to work correctly in a JSP 2.0 environment.

The following is a list of known backwards compatibility issues JSP developers should be aware of:

1. Tag Library Validators that are not namespace aware and that rely solely on the prefix parameter may not correctly validate some JSP 2.0 pages. This is because the XML view may contain tag library declarations in elements other than jsp:root, and may contain the same tag library declaration more than once, using different prefixes. The uri parameter should always be used by tag library validators instead. Existing JSP pages with existing tag libraries will not have any problems.

2. Users may observe differences in I18N behavior on some containers due primarily to ambiguity in the JSP 1.2 specification. Where possible, steps were taken to minimize the impact on backwards compatibility and overall, JSP's I18N abilities have been greatly improved.

   In JSP specification versions previous to JSP 2.0, JSP pages in XML syntax ("JSP documents") and those in standard syntax determined their page encoding in the same fashion, by examining the pageEncoding or contentType attributes of their page directive, defaulting to ISO-8859-1 if neither was present.

   As of JSP 2.0, the page encoding for JSP documents is determined as described in section 4.3.3 and appendix F.1 of the XML specification, and the pageEncoding attribute of those pages is only checked to make sure it is consistent with the page encoding determined as per the XML specification.

   As a result of this change, JSP documents that rely on their page encoding to be determined from their pageEncoding attribute will no longer be decoded correctly. These JSP documents must be changed to include an appropriate XML encoding declaration.

   Additionally, in JSP 1.2, page encodings are determined on a per translation unit basis whereas in JSP 2.0, page encodings are determined on a per-file basis. Therefore, if a.jsp statically includes b.jsp, and a page encoding is specified in a.jsp but not in b.jsp, in JSP 1.2 a.jsp's encoding is used for b.jsp, but in JSP 2.0, the default encoding is used for b.jsp.

3. The type coercion rules in Table JSP.1-11 have been reconciled with the EL coercion rules. There are some exceptional conditions that will no longer result in an exception in JSP 2.0. In particular, when passing an empty String("") to an attribute of a numeric type, a translation error or a NumberFormatException used to occur, whereas in JSP 2.0 a 0 will be passed in instead. See the

new Table JSP.1-11 for details. In general, this is not expected to cause any problems because these would have been exceptional conditions in JSP 1.2 and the specification allowed for these exceptions to occur at either translation time or request time.

The JSP container uses the version of web.xml to determine whether you are running a JSP 1.2 application or a JSP 2.0 application. Various features may behave differently depending on the version of web.xml. The following is a list of things JSP developers should be aware of when upgrading their web.xml from version Servlet 2.3 to version Servlet 2.4:

1. EL expressions will be ignored by default in JSP 1.2 applications. When upgrading a web application to JSP 2.0, EL expressions will be interpreted by default. The escape sequence \$ can be used to escape EL expressions that should not be interpreted by the container. Alternatively, the isELIgnored page directive attribute, or the <el-ignored> configuration element can be used to deactivate EL for entire translation units. Users of JSTL 1.0 will need to either upgrade their taglib imports to the JSTL 1.1 uris, or they will need to use the _rt versions of the tags (e.g. c_rt instead of c, or fmt_rt instead of fmt).
2. Web applications that contain files with an extension of .jspx will have those files interpreted as JSP documents, by default. You can use the JSP configuration element <is-xml> to treat .jspx files as regular JSP pages, but there is no way to disassociate .jspx from the JSP container.
3. The escape sequence \$ was not reserved in JSP 1.2. Any template text or attribute value that appeared as \$ in JSP 1.2 used to output \$ but will now output just $.

Licensing of Specification

Details on the conditions under which this document is distributed are described in the license agreement on page iii.

Who Should Read This Document

This document is the authoritative JSP 2.0 specification. It is intended to provide requirements for implementations of JSP page processing, and support by web containers in web servers and application servers. As an authoritative document, it

covers material pertaining to a wide audience, including *Page Authors*, *Tag Library Developers*, *Deployers*, *Container Vendors*, and *Tool Vendors*.

This document is not intended to be a user's guide. We expect other documents will be created that will cater to different readerships.

Organization of This Document

This document comprises of a number of Chapters and Appendices that are organized into 3 parts. In addition, the document contains a "Preface" (this section), a "Status" on page xxvii, and an "Overview" on page xxix.

Part I contains several chapters intended for all *JSP Page Authors*. These chapters describe the general structure of the language, including the expression language, fragments, and scripting.

Part II contains detailed chapters on the JSP container engine and API in full detail. The information in this part is intended for advanced JSP users.

Finally, Part III contains all the appendices.

Related Documents

Implementors of JSP containers and authors of JSP pages may find the following documents worth consulting for additional information

Table JSP.P-1 *Some Related Web Sites*

JSP home page	http://java.sun.com/products/jsp
Servlet home page	http://java.sun.com/products/servlet
Java 2 Platform, Standard Edition	http://java.sun.com/products/jdk/1.3 http://java.sun.com/products/jdk/1.4
Java 2 Platform, Enterprise Edition	http://java.sun.com/j2ee
XML in the Java Platform home page	http://java.sun.com/xml
JavaBeans™ technology home page	http://java.sun.com/beans
XML home page at W3C	http://www.w3.org/XML
HTML home page at W3C	http://www.w3.org/MarkUp
XML.org home page	http://www.xml.org

Table JSP.P-1 *Some Related Web Sites*

JSP home page	http://java.sun.com/products/jsp
Servlet home page	http://java.sun.com/products/servlet
JSR-045 home page (Debugging Support for Other Languages)	http://jcp.org/jsr/detail/45.jsp

Historical Note

The following individuals were pioneers who did ground-breaking work on the Java platform areas related to this specification. James Gosling's work on a Web Server in Java in 1994/1995 became the foundation for servlets. A larger project emerged in 1996 with Pavani Diwanji as lead engineer and with many other key members listed below. From this project came Sun's Java Web Server product.

Things started to move quickly in 1999. The servlet expert group, with James Davidson as lead, delivered the Servlet 2.1 specification in January and the Servlet 2.2 specification in December, while the JSP group, with Larry Cable and Eduardo Pelegri-Llopart as leads, delivered JSP 1.0 in June and JSP 1.1 in December.

The year 2000 saw a lot of activity, with many implementations of containers, tools, books, and training that target JSP 1.1, Servlet 2.2, and the Java 2 Platform, Enterprise Edition. Tag libraries were an area of intense development, as were varying approaches to organizing all these features together. The adoption of JSP technology has continued in the year 2001, with many talks at the "Web, Services and beyond" track at JavaOne being dedicated to the technology.

The JSP 1.2 specification went final in 2001. JSP 1.2 provided a number of fine-tunings of the spec. It also added the ability for validating JSP pages through the XML views of a JSP page. JSP 1.2 also introduced a normative XML syntax for JSP pages, but its adoption was handicaped by several specification shortcomings.

JSP 2.0 is a major revision of the JSP language. Key new features include a simple Expression Language, tag files, substantial simplifications for writing tag handlers in Java and the notion of JSP fragments. JSP 2.0 also includes a revision of the XML syntax that addresses most of the problems in JSP 1.2.

Tracking the industry in a printed document is at best difficult; the industry pages at the web site at http://java.sun.com/products/jsp do a better job.

Acknowledgments

Many people contributed to the JavaServer Pages specifications. The success of the Java Platform depends on the Java Community Process used to define and evolve it. This process, which involves many individuals and corporations, promotes the development of high quality specifications in Internet time.

Although it is impossible to list all the individuals who have contributed to this version of the specification, we would like to give thanks to all the members in our expert group. We have the benefit of a very large, active and enthusiastic expert group, without which the JSP specifications would not have succeeded.

We want to thank:

Nathan Abramson (Individual), Tim Ampe (Persistence Software Inc.), Shawn Bayern (Individual), Hans Bergsten (Individual), Paul Bonfanti (New Atlanta Communications Inc.), Prasad BV (Pramati Technologies), Bjorn Carlson (America Online), Murthy Chintalapati (Sun Microsystems, Inc.), Kin-Man Chung (Sun Microsystems, Inc.), Bill de hOra (InterX PLC), Ciaran Dynes (IONA Technologies PLC), Jayson Falkner (Individual), James Goodwill (Individual), Kouros Gorgani (Sybase), Randal Hanford (Boeing), Larry Isaacs (SAS Institute Inc.), Kevin R. Jones (Developmentor), Francois Jouaux (Apple Computer Inc.), Vishy Kasar (Borland Software Corporation), Ana Von Klopp (Sun Microsystems, Inc.), Matt LaMantia (Art Technology Group, Inc.), Bart Leeten (EDS), Geir Magnusson Jr. (Apache Software Foundation), Jason McGee (IBM), Brian McKellar (SAP AG), Shawn McMurdo (Lutris Technologies), Charles Morehead (Art Technology Group Inc.), Lars Oleson (SeeBeyond Technology Corp.), Jeff Plager (Sybase), Boris Pruessmann (Adobe Systems, Inc.), Tom Reilly (Macromedia, Inc.), Ricardo Rocha (Apache Software Foundation), John Rousseau (Novell, Inc.), James Strachan (Individual), Srinagesh Susarla (BEA Systems), Alex Yiu (Oracle).

We want to thank the community that implemented the reference implementation, and the vendors that have implemented the spec, the authoring tools, and the tag libraries.

Special mention is due to: Hans Bergsten for his numerous thorough reviews and technical accuracy, Shawn Bayern for his tireless help with the EL and RI, Alex Yiu for his thorough analysis on the invocation protocol and I18N, Nathan Abramson for his in-depth technical expertise and ideas, Norbert Lindenberg for his overhaul of the I18N chapter, Jan Luehe and Kin-Man Chung for keeping the RI more than up-to-date with the specification allowing for real-time feedback, Ana von Klopp for her help with JSR-45 debugging and keeping the tools

perspective fresh in our minds, and Umit Yalcinalp for her conversion of the TLD and deployment descriptors into XML Schema.

We want to thank all the authors of books on JSP technology, and the creators of the web sites that are tracking and facilitating the creation of the JSP community.

The editors want to give special thanks to many individuals within the Java 2 Enterprise Edition team, and especially to Jean-Francois Arcand, Jennifer Ball, Stephanie Bodoff, Pierre Delisle, Jim Driscoll, Cheng Fang, Robert Field, Justyna Horwat, Dianne Jiao, Norbert Lindenberg, Ryan Lubke, Jan Luehe, Craig McClanahan, Bill Shannon, Prasad Subramanian, Norman Walsh, Yutaka Yoshida, Kathleen Zelony, and to Ian Evans for his editorial work.

Lastly, but most importantly, we thank the software developers, web authors and members of the general public who have read this specification, used the reference implementation, and shared their experience. You are the reason the JavaServer Pages technology exists!

Status

This is the final draft of the JSP 2.0 specification, developed by the expert group JSR-152 under the Java Community Process (more details at http://jcp.org/jsr/detail/152.jsp).

The original Java Specification Request (JSR-152) listed the version number of the specification as 1.3. The scope and content of the specification effort has not changed, but the expert group realized that the new features would have a deep impact on the development model of JSP applications and decided that 2.0 would more appropriately reflect that impact.

The Java Community Process

The JCP produces a specification using three *communities*: an expert community (the *expert group*), the *participants* of the JCP, and the *public*-at-large. The expert group is responsible for the authoring of the specification through a collection of drafts. Specification *drafts* move from the expert community, through the participants, to the public, gaining in detail and completeness, always feeding received comments back to the expert group. The *final draft* is submitted for approval by the *Executive Committee.* The *expert group lead* is responsible for facilitating the workings of the expert group, for authoring the specification, and for delivering the *reference implementation* and the *conformance test suite*.

The JCP and This Specification

The JCP is designed to be a very flexible process so each expert group can address the requirements of the specific communities it serves. The reference imple-

mentation for JSP 2.0 and Servlet 2.4 uses code that is being developed as an open source project under an agreement with the Apache Software Foundation.

This specification includes chapters that are derived directly from the *Javadoc* comments in the API classes, but, were there to be any discrepancies, this specification has precedence over the Javadoc comments.

The JCP process provides a mechanism for updating the specification through a *maintenance* process using *erratas*. If available, the erratas will have precedence over this specification.

Overview

This is an overview of the JavaServer Pages technology.

The JavaServer Pages™ Technology

JavaServer™ Pages (JSP) is the Java™ 2 Platform, Enterprise Edition (J2EE) technology for building applications for generating dynamic web content, such as HTML, DHTML, XHTML, and XML. JSP technology enables the easy authoring of web pages that create dynamic content with maximum power and flexibility.

General Concepts

JSP technology provides the means for textual specification of the creation of a dynamic *response* to a *request*. The technology builds on the following concepts:

- *Template Data*

  A substantial portion of most dynamic content is fixed or *template* content. Text or XML fragments are typical template data. JSP technology supports natural manipulation of template data.

- *Addition of Dynamic Data*

  JSP technology provides a simple, yet powerful, way to add dynamic data to template data.

- *Encapsulation of Functionality*

  JSP technology provides two related mechanisms for the encapsulation of functionality: JavaBeans™ component architecture, and tag libraries deliver-

ing custom actions, functions, listener classes, and validation.

- *Good Tool Support*

  Good tool support leads to significantly improved productivity. Accordingly, JSP technology has features that enable the creation of good authoring tools.

  Careful development of these concepts yields a flexible and powerful server-side technology.

Benefits of JavaServer Pages Technology

JSP technology offers the following benefits:

- *Write Once, Run Anywhere™ properties*

  JSP technology is platform independent in its dynamic web pages, its web servers, and its underlying server components. JSP pages may be authored on any platform, run on any web server or web enabled application server, and accessed from any web browser. Server components can be built on any platform and run on any server.

- *High quality tool support*

  Platform independence allows the JSP user to choose best-of-breed tools. Additionally, an explicit goal of the JavaServer Pages design is to enable the creation of high quality portable tools.

- *Separation of Roles*

  JSP supports the separation of developer and author roles. *Developers* write components that interact with server-side objects. *Authors* put static data and dynamic content together to create presentations suited for their intended audience.
  Each group may do their job without knowing the job of the other. Each role emphasizes different abilities and, although these abilities may be present in the same individual, they most commonly will not be. Separation allows a natural division of labor.
  A subset of the developer community may be engaged in developing reusable components intended to be used by authors.

- *Reuse of components and tag libraries*

  JavaServer Pages technology emphasizes the use of reusable components

such as JavaBeans components, Enterprise JavaBeans™ components, and tag libraries. These components can be used with interactive tools for component development and page composition, yielding considerable development time savings. In addition, they provide the cross-platform power and flexibility of the Java programming language or other scripting languages.

- *Separation of dynamic and static content*

  JavaServer Pages technology enables the separation of static content in a template from dynamic content that is inserted into the static template. This greatly simplifies the creation of content. The separation is supported by beans specifically designed for the interaction with server-side objects, and by the tag extension mechanism.

- *Support for actions, expressions, and scripting*

  JavaServer Pages technology supports scripting elements as well as actions. Actions encapsulate useful functionality in a convenient form that can be manipulated by tools. Expressions are used to access data. Scripts can be used to glue together this functionality in a per-page manner.
  The JSP 2.0 specification adds a simple expression language (EL) to Java-based scripts. Expressions in the EL directly express page author concepts like properties in beans and provide more controlled access to the Web Application data. Functions defined through the tag library mechanism can be accessed in the EL.
  The JSP 2.0 specification also adds a mechanism by which page authors can write actions using the JSP technology directly. This greatly increases the ease with which action abstractions can be created.

- *Web access layer for N-tier enterprise application architecture(s)*

  JavaServer Pages technology is an integral part of J2EE. The J2EE platform brings Java technology to enterprise computing. One can now develop powerful middle-tier server applications that include a web site using JavaServer Pages technology as a front end to Enterprise JavaBeans components in a J2EE compliant environment.

Basic Concepts

This section introduces basic concepts that will be defined formally later in the specification.

What Is a JSP Page?

A JSP page is a text-based document that describes how to process a *request* to create a *response*. The description intermixes template data with dynamic actions and leverages the Java 2 Platform. JSP technology supports a number of different paradigms for authoring dynamic content. The key features of JavaServer Pages are:

- Standard directives
- Standard actions
- Scripting elements
- Tag Extension mechanism
- Template content

Web Applications

The concept of a web application is inherited from the servlet specification. A web application can be composed of:

- Java Runtime Environment(s) running on the server (required)
- JSP page(s) that handle requests and generate dynamic content
- Servlet(s) that handle requests and generate dynamic content
- Server-side JavaBeans components that encapsulate behavior and state
- Static HTML, DHTML, XHTML, XML, and similar pages.
- Client-side Java Applets, JavaBeans components, and arbitrary Java class files
- Java Runtime Environment(s) running in client(s) (downloadable via the Plugin and Java™ Web Start technology)

The JavaServer Pages specification inherits from the servlet specification the concepts of web applications, ServletContexts, sessions, and requests and responses. See the Java Servlet 2.4 specification for more details.

Components and Containers

JSP pages and servlet classes are collectively referred to as *web components*. JSP pages are delivered to a *container* that provides the services indicated in the *JSP Component Contract*.

The separation of components from containers allows the reuse of components, with quality-of-service features provided by the container.

Translation and Execution Steps

JSP pages are textual components. They go through two phases: a *translation* phase, and a *request* phase. Translation is done once per page. The request phase is done once per request.

The JSP page is translated to create a servlet class, the JSP page implementation class, that is instantiated at request time. The instantiated JSP page object handles requests and creates responses.

JSP pages may be translated prior to their use, providing the web application, with a servlet class that can serve as the textual representation of the JSP page.

The translation may also be done by the JSP container at deployment time, or on-demand as the requests reach an untranslated JSP page.

Deployment Descriptor and Global Information

The JSP pages delivered in a web application may require some JSP configuration information. This information is delivered through JSP-specific elements in the web.xml deployment descriptor, rooted on the <jsp-config> element. Configuration information includes <taglib> elements in mapping of tag libraries and <jsp-property-group> elements used to provide properties of collections of JSP files. The properties that can be indicated this way include page encoding information, EL evaluation activation, automatic includes before and after pages, and whether scripting is enabled in a given page.

Role in the Java 2 Platform, Enterprise Edition

With a few exceptions, integration of JSP pages within the J2EE 1.4 platform is inherited from the Servlet 2.4 specification since translation turns JSPs into servlets.

Users of JavaServer Pages

There are six classes of users that interact with JavaServer Pages technology. This section describes each class of user, enumerates the technologies each must be familiar with, and identifies which sections of this specification are most relevant to each user class. The intent is to ensure that JavaServer Pages remains a practical and

easy-to-use technology for each class of user, even as the language continues to grow.

Page Authors

Page Authors are application component providers that use JavaServer Pages to develop the presentation component of a web application. It is expected that they will not make use of the scripting capabilities of JavaServer Pages, but rather limit their use to standard and custom actions. Therefore, it is assumed that they know the target language, such as HTML or XML, and basic XML concepts, but they need not know Java at all.

The following sections are most relevant to this class of user:

- Chapter JSP.1, “Core Syntax and Semantics”, except for Section JSP.1.12, “Scripting Elements” and Section JSP.1.14, “Tag Attribute Interpretation Semantics”, which both talk about scripting.
- Chapter JSP.2, “Expression Language”
- Chapter JSP.3, “JSP Configuration”
- Chapter JSP.4, “Internationalization Issues”
- Chapter JSP.5, “Standard Actions”
- Chapter JSP.6, “JSP Documents”, except for sections that discuss declarations, scriptlets, expressions, and request-time attributes.
- Section JSP.7.1.1, “Goals” and Section JSP.7.1.2, “Overview” of Chapter JSP.7, “Tag Extensions”.
- Chapter JSP.8, “Tag Files”.
- Appendices JSP.A, JSP.E, and JSP.F.

Advanced Page Authors

Like Page Authors, Advanced Page Authors are also application component providers that use JavaServer Pages to develop the presentation component of a web application. These authors have a better understanding of XML and also know Java. Though they are recommended to avoid it where possible, these authors do have scripting at their disposal and should be able to read and understand JSPs that make use of scripting.

The following sections are most relevant to this class of user:

- Chapters JSP.1, JSP.2, JSP.3, JSP.4 and JSP.5.
- Chapter JSP.6, “JSP Documents”.
- Section JSP.9.1.1, “Valid JSP Page” and Section JSP.9.1.2, “Reserved Names” of Chapter JSP.9, “Scripting”.
- Section JSP.7.1.1, “Goals” and Section JSP.7.1.2, “Overview” of Chapter JSP.7, “Tag Extensions”.
- Chapter JSP.8, “Tag Files”
- Section JSP.11.4, “Precompilation” of Chapter JSP.11, “JSP Container”
- Chapter JSP.12, “Core API”
- Appendices JSP.A, JSP.B, JSP.E, and JSP.F.

Tag Library Developers

Tag Library Developers are application component providers who write tag libraries that provide increased functionality to Page Authors and Advanced Page Authors. They have an advanced understanding of the target language, XML, and Java.

The following sections are most relevant to this class of user:

- Chapters JSP.1, JSP.2, JSP.3, JSP.4 and JSP.5.
- Chapter JSP.6, “JSP Documents”.
- Section JSP.9.1.1, “Valid JSP Page” and Section JSP.9.1.2, “Reserved Names” of Chapter JSP.9, “Scripting”.
- Chapter JSP.7, “Tag Extensions”
- Chapter JSP.8, “Tag Files”
- Section JSP.11.4, “Precompilation” of Chapter JSP.11, “JSP Container”
- Chapter JSP.12, “Core API” and Chapter JSP.13, “Tag Extension API”
- All Appendices.

Deployers

A deployer is an expert in a specific operational environment who is responsible for configuring a web application for, and deploying the web application to, that environment. The deployer does not need to understand the target language or Java,

but must have an understanding of XML or use tools that provide the ability to read deployment descriptors.

The following sections are most relevant to this class of user:

- Section JSP.1.1, “What Is a JSP Page” and Section JSP.1.2, “Web Applications” of Chapter JSP.1, “Core Syntax and Semantics”
- Chapter JSP.3, “JSP Configuration”
- Chapter JSP.4, “Internationalization Issues”
- Chapter JSP.11, “JSP Container”
- All Appendices.

Container Developers and Tool Vendors

Container Developers develop containers that host JavaServer Pages. Tool Vendors write development tools to assist Page Authors, Advanced Page Authors, Tag Library Developers, and Deployers. Both Container Developers and Tool Vendors must know XML and Java, and must know all the requirements and technical details of JavaServer Pages. Therefore, this entire specification is relevant to both classes of user.

Part I

The next chapters form the core of the JSP specification. These chapters provide information for Page authors, Tag Library developers, deployers and Container and Tool vendors.

The chapter of this part are

- Core Syntax and Semantics
- Expression Language
- Configuration Information
- Internationalization Issues
- Standard Actions
- JSP Documents
- Tag Extensions
- Tag Files
- Scripting
- XML Views

CHAPTER JSP.1

Core Syntax and Semantics

This chapter describes the core syntax and semantics for the JavaServer Pages 2.0 specification (JSP 2.0).

JSP.1.1 What Is a JSP Page

A JSP page is a textual document that describes how to create a response object from a request object for a given protocol. The processing of the JSP page may involve creating and/or using other objects.

A JSP page defines a JSP page implementation class that implements the semantics of the JSP page. This class implements the javax.servlet.Servlet interface (see Chapter JSP.11 for details). At request time a request intended for the JSP page is delivered to the JSP page implementation object for processing.

HTTP is the default protocol for requests and responses. Additional request/response protocols may be supported by JSP containers. The default request and response objects are of type HttpServletRequest and HttpServletResponse respectively.

JSP.1.1.1 Web Containers and Web Components

A JSP container is a system-level entity that provides life-cycle management and runtime support for JSP pages and servlet components. Requests sent to a JSP page are delivered by the JSP container to the appropriate JSP page implementation object. The term web container is synonymous with JSP container.

A web component is either a servlet or a JSP page. The servlet element in a web.xml deployment descriptor is used to describe both types of web components. JSP page components are defined implicitly in the deployment descriptor through

the use of an implicit .jsp extension mapping, or explicitly through the use of a jsp-group element.

JSP.1.1.2 Generating HTML

A traditional application domain of the JSP technology is HTML content. The JSP specification supports well this use through a syntax that is friendly to HTML and XML although it is not HTML-specific; for instance, HTML comments are treated no differently than other HTML content. The JSP Standard Tag Library has specific support for HTML though some specific custom actions.

JSP.1.1.3 Generating XML

An increasingly important application domain for JSP technology is dynamic XML content using formats like XHTML, SVG and the Open Office format, and in applications like content publishing, data representation and Web Services. The basic JSP machinery (JSP syntax) can be used to generate XML content, but it is also possible to tag a JSP page as a JSP document and get additional benefits.

A JSP document is an XML document; this means that a JSP document is a well-formed, structured document and that this will be validated by the JSP container. Additionally, this structure will be available to the JSP validation machinery, the TagLibraryValidators. A JSP document is a namespace-aware XML document, with namespaces reflecting the structure of both content and custom actions and with some additional care, a JSP page can reflect quite accurately the structure of the resulting content. A JSP document can also use machinery like entity definitions.

The JSP 1.2 specification made a stronger distinction between JSP documents and non-XML JSP pages. For instance standard actions like <jsp:expression> were only available in JSP documents. The difference proved to be confusing and distracting and the distinction has been relaxed in JSP 2.0 to facilitate the transition from the JSP syntax to XML syntax.

JSP.1.1.4 Translation and Execution Phases

A JSP container manages two phases of a JSP page's lifecycle. In the *translation phase*, the container validates the syntactic correctness of the JSP pages and tag files and determines a JSP page implementation class that corresponds to the JSP page. In the *execution phase* the container manages one or more instances of this class in response to requests and other events.

During the translation phase the container locates or creates the JSP page implementation class that corresponds to a given JSP page. This process is determined by the semantics of the JSP page. The container interprets the standard directives and actions, and the custom actions referencing tag libraries used in the page. A tag library may optionally provide a validation method acting on the XML View of a JSP page, see below, to validate that a JSP page is correctly using the library.

A JSP container has flexibility in the details of the JSP page implementation class that can be used to address quality-of-service--most notably performance--issues.

During the execution phase the JSP container delivers events to the JSP page implementation object. The container is responsible for instantiating request and response objects and invoking the appropriate JSP page implementation object. Upon completion of processing, the response object is received by the container for communication to the client. The details of the contract between the JSP page implementation class and the JSP container are described in Chapter JSP.11.

The translation of a JSP source page into its implementation class can occur at any time between initial deployment of the JSP page into the JSP container and the receipt and processing of a client request for the target JSP page. Section JSP.1.1.9 describes how to perform the translation phase ahead of deployment.

JSP.1.1.5 Validating JSP pages

All JSP pages, regardless of whether they are written in the traditional JSP syntax or the XML syntax of JSP documents have an equivalent XML document, the XML view of a JSP page, that is presented to tag library validators in the translation phase for validation.

The structure of the custom actions in a JSP page is always exposed in the XML view. This means that a tag library validator can check that, for instance, some custom actions are only used within others.

The structure of the content used in a JSP page is exposed in greater or lesser detail depending on whether the XML syntax or the traditional JSP syntax is used. When using XML syntax a tag library validator can use that extra structure to, for example, check that some actions are only used with some content, or within some content, and, using knowledge of the semantics of the custom actions, make assertions on the generated dynamic content.

JSP.1.1.6 Events in JSP Pages

A JSP page may indicate how some events are to be handled.

As of JSP 1.2 only init and destroy events can be described in the JSP page. When the first request is delivered to a JSP page, a jspInit() method, if present, will be called to prepare the page. Similarly, a JSP container invokes a JSP's jspDestroy() method to reclaim the resources used by the JSP page at any time when a request is not being serviced. This is the same life-cycle as for servlets.

JSP.1.1.7 JSP Configuration Information

JSP pages may be extended with configuration information that is delivered in the JSP configuration portion of the web.xml deployment description of the web application. The JSP configuration information includes interpretation for the tag libraries used in the JSP files and different property information for groups of JSP files. The property information includes: page encoding information, whether the EL evaluation and the scripting machinery is enabled, and prelude and coda automatic inclusions. The JSP configuration information can also be used to indicate that some resources in the web application are JSP files even if they do not conform to the default .jsp extension, and to modify the default interpretation for .jspx.

JSP.1.1.8 Naming Conventions for JSP Files

A JSP page is packaged as one or more JSP files, often in a web application, and delivered to a tool like a JSP container, a J2EE container, or an IDE. A complete JSP page may be contained in a single file. In other cases, the top file will include other files that contain complete JSP pages, or included segments of pages.

It is common for tools to need to differentiate JSP files from other files. In some cases, the tools also need to differentiate between top JSP files and included segments. For example, a segment may not be a legal JSP page and may not compile properly. Determining the type of file is also very useful from a documentation and maintenance point of view, as people familiar with the .c and .h convention in the C language know.

By default the extension .jsp means a top-level JSP file. We recommend, but do not mandate, to differentiate between top-level JSP files (invoked directly by the client or dynamically included by another page or servlet) and statically included segments so that:

- The .jsp extension is used only for files corresponding to top level JSP files, forming a JSP page when processed.
- Statically included segments use any other extension. As included segments were called ‘JSP fragments’ in past versions of this specification, the extension .jspf was offered as a suggestion. This extension is still suggested for consistency reasons, despite that they are now called ‘jsp segments’.

JSP documents, that is, JSP pages that are delivered as XML documents, use the extension .jspx by default.

The jsp-property-group element of web.xml can be used to indicate that some group of files, perhaps not using either of the extensions above, are JSP pages, and can also be used to indicate which ones are delivered as XML documents.

JSP.1.1.9 Compiling JSP Pages

A JSP page may be compiled into its implementation class plus deployment information during development (a JSP page can also be compiled at deployment time). In this way JSP page authoring tools and JSP tag libraries may be used for authoring servlets. The benefits of this approach include:

- Removal of the start-up lag that occurs when a container must translate a JSP page upon receipt of the first request.
- Reduction of the footprint needed to run a JSP container, as the Java compiler is not needed.

Compilation of a JSP page in the context of a web application provides resolution of relative URL specifications in include directives and elsewhere, tag library references, and translation-time actions used in custom actions.

A JSP page can also be compiled at deployment time.

JSP.1.1.9.1 JSP Page Packaging

When a JSP page implementation class depends on support classes in addition to the JSP 2.0 and Servlet 2.4 classes, the support classes are included in the packaged WAR, as defined in the Servlet 2.4 specification, for portability across JSP containers.

Appendix JSP.A contains two examples of JSP pages packaged in WARs:

1. A JSP page delivered in source form (the most common case).

2. A JSP page translated into an implementation class plus deployment information. The deployment information indicates support classes needed and the mapping between the original URL path to the JSP page and the URL for the JSP page implementation class for that page.

JSP.1.1.10 Debugging JSP Pages

In the past debugging tools provided by development environments have lacked a standard format for conveying source map information allowing the debugger of one vendor to be used with the JSP container of another. As of JSP 2.0, containers must support JSR-045 (“Debugging Support for Other Languages”). Details can be found in Section JSP.11.5, “Debugging Requirements”.

JSP.1.2 Web Applications

A web application is a collection of resources that are available at designated URLs. A web application is made up of some of the following:

- Java runtime environment(s) running in the server (required)
- JSP page(s) that handle requests and generate dynamic content
- Servlet(s) that handle requests and generate dynamic content
- Server-side JavaBeans components that encapsulate behavior and state
- Static HTML, DHTML, XHTML, XML and similar pages.
- Resource files used by Java classes.
- Client-side Java Applets, JavaBeans components, and Java class files
- Java runtime environment(s) (downloadable via the Plugin and Java Web Start) running in client(s)

Web applications are described in more detail in the Servlet 2.4 specification.

A web application contains a deployment descriptor web.xml that contains information about the JSP pages, servlets, and other resources used in the web application. The deployment descriptor is described in detail in the Servlet 2.4 specification.

JSP 2.0 requires that these resources be implicitly associated with and accessible through a unique ServletContext instance available as the implicit application object (see Section JSP.1.8).

The application to which a JSP page belongs is reflected in the application object, and has impact on the semantics of the following elements:

- The include directive (see Section JSP.1.10.3).
- The taglib directive (see Section JSP.1.10.2).
- The jsp:include action element (see Section JSP.5.4).
- The jsp:forward action (see Section JSP.5.5).

JSP 2.0 supports portable packaging and deployment of web applications through the Servlet 2.4 specification. The JavaServer Pages specification inherits from the servlet specification the concepts of applications, ServletContexts, Sessions, Requests and Responses.

JSP.1.2.1 Relative URL Specifications

Elements may use relative URL specifications, called URI paths, in the Servlet 2.4 specification. These paths are as described in RFC 2396. We refer to the path part of that specification, not the scheme, nor authority parts. Some examples are:

- A context-relative path is a path that starts with a slash (/). It is to be interpreted as relative to the application to which the JSP page or tag file belongs. That is, its ServletContext object provides the base context URL.
- A page relative path is a path that does not start with a slash (/). It is to be interpreted as relative to the current JSP page, or the current JSP file or tag file, depending on where the path is being used. For an include directive (see Section JSP.1.10.3) where the path is used in a file attribute, the interpretation is relative to the JSP file or tag file. For a jsp:include action (see Section JSP.5.4) where the path is used in a page attribute, the interpretation is relative to the JSP page. In both cases the current page or file is denoted by some path starting with / that is then modified by the new specification to produce a path starting with /. The new path is interpreted through the ServletContext object. See Section JSP.1.10.5 for exact details on this interpretation.

The JSP specification uniformly interprets paths in the context of the web container where the JSP page is deployed. The specification goes through a mapping translation. The semantics outlined here apply to the translation-time phase, and to the request-time phase.

JSP.1.3 Syntactic Elements of a JSP Page

This section describes the basic syntax rules of JSP pages.

JSP.1.3.1 Elements and Template Data

A JSP page has elements and template data. An element is an instance of an element type known to the JSP container. Template data is everything else; that is, anything that the JSP translator does not know about.

The type of an element describes its syntax and its semantics. If the element has attributes, the type describes the attribute names, their valid types, and their interpretation. If the element defines objects, the semantics includes what objects it defines and their types.

JSP.1.3.2 Element Syntax

There are three types of elements: directive elements, scripting elements, and action elements.

Directives

Directives provide global information that is conceptually valid independent of any specific request received by the JSP page. They provide information for the translation phase.

Directive elements have a syntax of the form <%@ directive...%>.

Actions

Actions provide information for the request processing phase. The interpretation of an action may, and often will, depend on the details of the specific request received by the JSP page. An Actions can either be *standard* (that is. defined in this specification), or *custom* (that is, provided via the portable tag extension mechanism).

Action elements follow the syntax of an XML element. They have a start tag including the element name, and may have attributes, an optional body, and a matching end tag, or may be an empty tag, possibly with attributes:

```
<mytag attr1=”attribute value”...>body</mytag>
```

And:

```
<mytag attr1="attribute value".../>
<mytag attr1="attribute value" ...></mytag>
```

An element has an *element type* describing its tag name, its valid attributes and its semantics. We refer to the type by its tag name.
JSP tags are case-sensitive, as in XML and XHTML.
An action may create objects and may make them available to the scripting elements through scripting-specific variables.

Scripting Elements

Scripting elements provide "glue" around template text and actions.
JSP 2.0 has a simple Expression Language (EL) that can be used to simplify accessing data from different sources. EL expressions can be used in JSP standard and custom actions and template data. EL expressions use the syntax ${expr}; For example:

```
<mytag attr1="${bean.property}".../>
${map[entry]}
<lib:myAction>${3+counter}</lib:myAction>
```

Chapter JSP.2 provides more details on the EL.

JSP 2.0 retains the three language-based types of scripting elements: *declarations*, *scriptlets*, and *expressions.* Declarations follow the syntax <%! ... %>. Scriptlets follow the syntax <% ... %>. Expressions follow the syntax <%= ... %>.

JSP.1.3.3 Start and End Tags

Elements that have distinct start and end tags (with enclosed body) must start and end in the same file. The start tag cannot be on one file while the end tag is in another.

The same rule applies to elements in the alternate syntax. For example, a scriptlet has the syntax <% scriptlet %>. Both the opening <% characters and the closing %> characters must be in the same physical file.

A scripting language may also impose constraints on the placement of start and end tags relative to specific scripting constructs. For example, Chapter 9 shows that Java language blocks cannot separate a start and an end tag. See Section JSP.9.4 for details.

JSP.1.3.4 Empty Elements

Following the XML specification, an element described using an empty tag is indistinguishable from one using a start tag, an empty body, and an end tag

As examples, the following are all empty tags:

```
<x:foo></x:foo>
<x:foo />
<x:foo/>
<x:foo><%-- any comment --%></x:foo>
```

While the following are all non-empty tags:

```
<foo> </foo>
<foo><%= expression %></foo>
<foo><% scriptlet %></foo>
<foo><bar/></foo>
<foo><!-- a comment --></foo>
```

JSP.1.3.5 Attribute Values

Following the XML specification, attribute values always appear quoted. Either single or double quotes can be used to reduce the need for escaping quotes; the quotation conventions available are described in Section JSP.1.6. There are two types of attribute values, literals and request-time expressions (Section JSP.1.14.1), but the quotation rules are the same.

JSP.1.3.6 The jsp:attribute, jsp:body and jsp:element Elements

Until JSP 2.0, tag handlers could be passed input two ways: through attribute values and through the element body. Attribute values were always evaluated once (if they were specified as an expression) and the result was passed to the tag handler. The body could contain scripting elements and action elements and be evaluated zero or more times on demand by the tag handler.

As of JSP 2.0, page authors can provide input in new ways using the <jsp:attribute> standard action element. Based on the configuration of the action being invoked, the body of the element either specifies a value that is evaluated once, or it specifies a “JSP fragment,” which represents the body in a form that makes it possible for a tag handler to evaluate it as many times as needed. The <jsp:attribute> action must only be used to specify an attribute value for standard or custom actions. A translation error must occur if it is used in any other context, for example to specify the value of template text that looks like an XML element.

It is illegal JSP syntax, which must result in a translation error, to use both an XML element attribute and a <jsp:attribute> standard action to pass the value of the same attribute. See Section JSP.5.10 for more details on the <jsp:attribute> standard action.

The following example uses an XML element attribute to define the value of the param1 attribute, and uses an attribute standard action to define the value of the param2 attribute. In this example, the value of param2 comes from the result of a custom action invocation.

```
<mytag:paramTag param1="value1">
    <jsp:attribute name="param2">
        <mymath:add x="2" y="2"/>
    </jsp:attribute>
</mytag:paramTag>
```

If a page author wishes to pass both an attribute standard action and a tag body, the <jsp:body> standard action must be used to specify the body. A translation error will result if the custom action invocation has <jsp:attribute> elements but does not define the body using a <jsp:body> element. See Section JSP.5.11 for more details on the <jsp:body> standard action.

The following example shows two equivalent tag invocations to the hypothetical <mytag:formatBody> custom action. The first invocation uses an XML element attribute to pass the values of the color and size attributes. The second example uses an attribute standard action to pass the value of the color attribute. Both examples have tag body containing simply the words "Template Text".

```
<mytag:tagWithBody color="blue" size="12">
    Template Text
</mytag:tagWithBody>

<mytag:tagWithBody size="12">
    <jsp:attribute name="color">blue</jsp:attribute>
    <jsp:body>
        Template Text
    </jsp:body>
</mytag:tagWithBody>
```

<jsp:attribute> can be used with the <jsp:element> standard action to generate dynamic content in a well structured way. The example below generates an HTML head of some type unknown at page authoring time:

```
<jsp:element name="H${headLevel}">
    <jsp:attribute name="size">${headSize}</jsp:attribute>
    <jsp:body>${headText}<jsp:body>
</jsp:element>
```

JSP.1.3.7 Valid Names for Actions and Attributes

The names for actions must follow the XML convention (i.e. must be an NMTOKEN as indicated in the XML 1.0 specification). The names for attributes must follow the conventions described in the JavaBeans specification.

Attribute names that start with jsp, _jsp, java, or sun are reserved in this specification.

JSP.1.3.8 White Space

In HTML and XML white space is usually not significant, but there are exceptions. For example, an XML file may start with the characters <?xml, and, when it does, it must do so with no leading whitespace characters.

This specification follows the whitespace behavior defined for XML. White space within the body text of a document is not significant, but is preserved.

Next are two examples of JSP code with their associated output. Note that directives generate no data and apply globally to the JSP page.

Table JSP.1-1 *Example 1 - Input*

LineNo	Source Text
1	<?xml version="1.0" ?>
2	<%@ page buffer="8kb" %>
3	The rest of the document goes here

The result is

Table JSP.1-2 *Example 1 - Output*

LineNo	Output Text
1	<?xml version="1.0" ?>
2	
3	The rest of the document goes here

The next two tables show another example, with input and output.,

Table JSP.1-3 *Example 2 - Input*

LineNo	Source Text
1	<% response.setContentType(“....”);
2	whatever... %><?xml version=”1.0” ?>
3	<%@ page buffer=”8kb” %>
4	The rest of the document goes here

The result is

Table JSP.1-4 *Example 2 - Output*

LineNo	Output Text
1	<?xml version=”1.0” ?>
2	
3	The rest of the document goes here

JSP.1.3.9 JSP Documents

A JSP page is usually passed directly to a JSP container. A JSP Document is a JSP page that is also an XML document. When a JSP document is encountered by the JSP container, it is interpreted as an XML document first and after that as a JSP page. Among the consequences of this are:

- The document must be well-formed
- Validation, if indicated
- Entity resolution will apply, if indicated
- <% style syntax cannot be used

JSP documents are often a good match for the generation of dynamic XML content as they can preserve much of the structure of the generated document.

The default convention for JSP documents is .jspx. There are configuration elements that can be used to indicate that a specific file is a JSP document.

See Chapter JSP.6 for more details on JSP documents, and Chapter 3 for more details on configuration.

JSP.1.3.10 JSP Syntax Grammar

This section presents a simple EBNF grammar for the JSP syntax. The grammar is intended to provide a concise syntax overview and to resolve any syntax ambiguities present in this specification. Other sections may apply further restrictions to this syntax, for example to restrict what represents a valid attribute value for a page directive. In all other cases the grammar takes precedence in resolving syntax questions.

The notation for this grammar is identical to that described by Chapter 6 of the XML 1.0 specification, available at the following URL:

http://www.w3c.org/TR/2000/REC-xml-20001006#sec-notation

In addition, the following notes and rules apply:

- The root production for a JSP page is JSPPage.
- The prefix XML:: is used to refer to an EBNF definition in the XML 1.0 specification. Refer to http://www.w3.org/TR/REC-xml.
- Where applicable, to resolve grammar ambiguities, the first matching production must always be followed. This is commonly known as the “greedy” algorithm.
- If the <TRANSLATION_ERROR> production is followed, the page is invalid, and the result will be a translation error.
- Many productions make use of XML-style attributes. These attributes can appear in any order, separated from each other by whitespace, but no attribute can be repeated more than once. To make these XML-style attribute specifications more concise and easier to read, the syntax ATTR[*attrset*] is used in the EBNF to define a set of XML attributes that are recognized in a particular production.

  Within the square brackets (*attrset*) is listed a comma-separated list of case-sensitive attribute names that are valid. Each attribute name represents a single XML attribute. If the attribute name is prefixed with an =, the production Attribute (defined below) must be matched (either a rtexprvalue or a static value is accepted). If not, the production NonRTAttribute must be matched (only static values are accepted). If the attribute name is prefixed with a !, the attribute is required and must appear in order for this production to be matched. If an attribute that matches the Attribute production with a name not listed appears adjacent to any of the other attributes, the production is not matched.

For example, consider a production that contains ATTR[!name, =value, =!repeat]. This production is matched if and only if all of the following hold true:

- The name attribute appears exactly once and matches the NonRTAttribute production.
- The value attribute appears at most once. If it appears, the Attribute production must be matched.
- The repeat attribute appears exactly once and matches the Attribute production.
- There must be no other attributes aside from name, value, or repeat.

For example, the following sample strings match the above:

- name="somename" value="somevalue" repeat="2"
- repeat="${ x + y }" name="othername"

JSP.1.3.10.1 EBNF Grammar for JSP Syntax

```
JSPPage         ::= Body

JSPTagDef       ::= Body

Body            ::= AllBody | ScriptlessBody
                    [ vc: ScriptingEnabled ]
                    [ vc: ScriptlessBody ]

AllBody         ::= (       ( '<%--'              JSPCommentBody   )
                        |   ( '<%@'               DirectiveBody    )
                        |   ( '<jsp:directive.'   XMLDirectiveBody )
                        |   ( '<%!'               DeclarationBody  )
                        |   ( '<jsp:declaration'  XMLDeclarationBody)
                        |   ( '<%='               ExpressionBody   )
                        |   ( '<jsp:expression'   XMLExpressionBody)
                        |   ( '<%'                ScriptletBody    )
                        |   ( '<jsp:scriptlet'    XMLScriptletBody )
                        |   ( '${'                ELExpressionBody )
                        |   ( '<jsp:text'         XMLTemplateText  )
                        |   ( '<jsp:'             StandardAction   )
                        |   ( '</'                ExtraClosingTag  )
                        |   ( '<'                 CustomAction
                                                  CustomActionBody )
                        |   TemplateText
                    )*
```

```
ScriptlessBody      ::= (       ( '<%--'                JSPCommentBody   )
                        |       ( '<%@'                 DirectiveBody    )
                        |       ( '<jsp:directive.'     XMLDirectiveBody )
                        |       ( '<%!'         <TRANSLATION_ERROR>      )
                        |       ( '<jsp:declaration'
                                                <TRANSLATION_ERROR>      )
                        |       ( '<%='         <TRANSLATION_ERROR>      )
                        |       ( '<jsp:expression'
                                                <TRANSLATION_ERROR>      )
                        |       ( '<%'          <TRANSLATION_ERROR>      )
                        |       ( '<jsp:scriptlet'
                                                <TRANSLATION_ERROR>      )
                        |       ( '${'                  ELExpressionBody )
                        |       ( '<jsp:text'           XMLTemplateText  )
                        |       ( '<jsp:'               StandardAction   )
                        |       ( '</'                  ExtraClosingTag  )
                        |       ( '<'                   CustomAction
                                                        CustomActionBody )
                        |       TemplateText
                        )*
                        [ vc: ELEnabled ]

TemplateTextBody    ::= (       ( '<%--'                JSPCommentBody   )
                        |       ( '<%@'                 DirectiveBody    )
                        |       ( '<jsp:directive.'     XMLDirectiveBody )
                        |       ( '<%!'         <TRANSLATION_ERROR>      )
                        |       ( '<jsp:declaration'
                                                <TRANSLATION_ERROR>      )
                        |       ( '<%='         <TRANSLATION_ERROR>      )
                        |       ( '<jsp:expression'
                                                <TRANSLATION_ERROR>      )
                        |       ( '<%'          <TRANSLATION_ERROR>      )
                        |       ( '<jsp:scriptlet'
                                                <TRANSLATION_ERROR>      )
                        |       ( '${'          <TRANSLATION_ERROR>      )
                        |       ( '<jsp:text'   <TRANSLATION_ERROR>      )
                        |       ( '<jsp:'       <TRANSLATION_ERROR>      )
                        |       ( '<'           CustomAction
                                                <TRANSLATION_ERROR>      )
                        |       TemplateText
                        )*
                        [ vc: ELEnabled ]
```

```
JSPCommentBody  ::=     ( Char* - ( Char* '--%>' ) ) '--%>'
                    |   <TRANSLATION_ERROR>

DirectiveBody       ::= JSPDirectiveBody | TagDefDirectiveBody
                        [ vc: TagFileSpecificDirectives ]

XMLDirectiveBody    ::= XMLJSPDirectiveBody | XMLTagDefDirectiveBody
                        [ vc: TagFileSpecificXMLDirectives ]

JSPDirectiveBody    ::=     S?
                            (       ( 'page' S PageDirectiveAttrList )
                                |   ( 'taglib' S TagLibDirectiveAttrList )
                                |   ( 'include' S IncludeDirectiveAttrList )
                            )
                            S? '%>'
                        |   <TRANSLATION_ERROR>

XMLJSPDirectiveBody::=      S?
                            (       (   'page' S PageDirectiveAttrList S?
                                        ( '/>' | ( '>' S? ETag ) )
                                    )
                                |   (   'include' S IncludeDirectiveAttrList S?
                                        ( '/>' | ( '>' S? ETag ) )
                                    )
                            )
                        |   <TRANSLATION_ERROR>

TagDefDirectiveBody::=      S?
                            (       ( 'tag'         S TagDirectiveAttrList )
                                |   ( 'taglib'      S TagLibDirectiveAttrList )
                                |   ( 'include'     S IncludeDirectiveAttrList )
                                |   ( 'attribute'   S AttributeDirectiveAttrList )
                                |   ( 'variable'    S VariableDirectiveAttrList )
                            )
                            S? '%>'
                        |   <TRANSLATION_ERROR>
```

```
XMLTagDefDirectiveBody::= (       (    'tag' S TagDirectiveAttrList S?
                                       ( '/>' | ( '>' S? ETag ) )
                                  )
                          |       (    'include' S IncludeDirectiveAttrList S?
                                       ( '/>' | ( '>' S? ETag ) )
                                  )
                          |       ( 'attribute' S AttributeDirectiveAttrList S?
                                       ( '/>' | ( '>' S? ETag ) )
                                  )
                          |       ( 'variable' S VariableDirectiveAttrList S?
                                       ( '/>' | ( '>' S? ETag ) )
                                  )
                          )
                    |     <TRANSLATION_ERROR>

PageDirectiveAttrList::=  ATTR[   language, extends, import, session,
                                  buffer, autoFlush, isThreadSafe,
                                  info, errorPage, isErrorPage,
                                  contentType, pageEncoding,
                                  isELIgnored                          ]
                          [ vc: PageDirectiveUniqueAttr ]

TagLibDirectiveAttrList::=      ATTR[ !uri, !prefix     ]
                          |     ATTR[ !tagdir, !prefix ]
                          [ vc: TagLibDirectiveUniquePrefix ]

IncludeDirectiveAttrList::=ATTR[  !file ]

TagDirectiveAttrList  ::= ATTR[   display-name, body-content,
                                  dynamic-attributes, small-icon, large-icon,
                                  description, example, language,
                                  import, pageEncoding, isELIgnored        ]
                          [ vc: TagDirectiveUniqueAttr ]

AttributeDirectiveAttrList::=ATTR[ !name, required, fragment, rtexprvalue,
                                  type, description ]
                          [ vc: UniqueAttributeName ]

VariableDirectiveAttrList::=    ATTR[   !name-given, variable-class,
                                        scope, declare, description ]
                          |     ATTR[   !name-from-attribute, !alias,
                                        variable-class,
                                        scope, declare, description ]
                          [ vc: UniqueVariableName ]
```

```
DeclarationBody      ::=    ( Char* - ( Char* '%>' ) ) '%>'
                        |   <TRANSLATION_ERROR>

XMLDeclarationBody::=       ( S? '/>' )
                        |   (   S? '>'
                                ( ( Char* - ( Char* '<' ) ) CDSect? )*
                                ETag
                            )
                        |   <TRANSLATION_ERROR>

ExpressionBody       ::=    ( Char* - ( Char* '%>' ) ) '%>'
                        |   <TRANSLATION_ERROR>
                        [ vc: ExpressionBodyContent ]

XMLExpressionBody::=        ( S? '/>' )
                        |   (   S? '>'
                                ( ( Char* - ( Char* '<' ) ) CDSect? )*
                                ETag
                            )
                        |   <TRANSLATION_ERROR>
                        [ vc: ExpressionBodyContent ]

ELExpressionBody     ::=    ELExpression '}'
                        |   <TRANSLATION_ERROR>

ELExpression         ::=  [See Section JSP.2.9, production Expression]

ScriptletBody        ::=    ( Char* - ( Char* '%>' ) ) '%>'
                        |   <TRANSLATION_ERROR>

XMLScriptletBody     ::=    ( S? '/>' )
                        |   (   S? '>'
                                ( ( Char* - ( Char* '<' ) ) CDSect? )*
                                ETag
                            )
                        |   <TRANSLATION_ERROR>
```

```
StandardAction      ::=     ( 'useBean'      StdActionContent )
                        |   ( 'setProperty'  StdActionContent )
                        |   ( 'getProperty'  StdActionContent )
                        |   ( 'include'      StdActionContent )
                        |   ( 'forward'      StdActionContent )
                        |   ( 'plugin'       StdActionContent )
                        |   ( 'invoke'       StdActionContent )
                        |   ( 'doBody'       StdActionContent )
                        |   ( 'element'      StdActionContent )
                        |   ( 'output'       StdActionContent )
                        |   <TRANSLATION_ERROR>
                        [ vc: TagFileSpecificActions ]

StdActionContent    ::= Attributes StdActionBody
                        [ vc: StdActionAttributesValid ]

StdActionBody       ::=     EmptyBody
                        |   OptionalBody
                        |   ParamBody
                        |   PluginBody
                        [ vc: StdActionBodyMatch ]

EmptyBody           ::=     '/>'
                        |   ( '>' ETag )
                        |   ( '>' S? '<jsp:attribute' NamedAttributes ETag )

TagDependentActionBody ::=  JspAttributeAndBody
                        |   ( '>' TagDependentBody ETag )

TagDependentBody    ::= Char* - ( Char* ETag )

JspAttributeAndBody::= (    '>' S? ( '<jsp:attribute'NamedAttributes )?
                            '<jsp:body'
                            ( JspBodyBody |<TRANSLATION_ERROR> )
                            S? ETag
                        )

ActionBody          ::=     JspAttributeAndBody
                        |   ( '>' Body ETag )

ScriptlessActionBody::=     JspAttributeAndBody
                        |   ( '>' ScriptlessBody ETag )

OptionalBody        ::= EmptyBody | ActionBody
```

```
ScriptlessOptionalBody::=EmptyBody | ScriptlessActionBody

TagDependentOptionalBody::= EmptyBody | TagDependentActionBody

ParamBody          ::=     EmptyBody
                     |     (     '>' S? ( '<jsp:attribute' NamedAttributes )?
                                 '<jsp:body'
                                 (JspBodyParam | <TRANSLATION_ERROR>)
                                 S? ETag
                           )
                     |     ( S? '>' Param* ETag )

PluginBody         ::=     EmptyBody
                     |     (     '>' S? ( '<jsp:attribute' NamedAttributes )?
                                 '<jsp:body'
                                 (           JspBodyPluginTags
                                       |     <TRANSLATION_ERROR>
                                 )
                                 S? ETag
                           )
                     |     ( '>' S? PluginTags ETag )

NamedAttributes    ::=  AttributeBody S? ( '<jsp:attribute' AttributeBody S? )*

AttributeBody      ::=     ATTR[   !name, trim  ] S?
                     (           '/>'
                           |     '></jsp:attribute>'
                           |     '>' AttributeBodyBody '</jsp:attribute>'
                           |     <TRANSLATION_ERROR>
                     )

AttributeBodyBody  ::=     AllBody
                     |     ScriptlessBody
                     |     TemplateTextBody
                     [ vc: AttributeBodyMatch ]

JspBodyBody        ::=     ( S? JspBodyEmptyBody )
                     |     ( S? '>' ( JspBodyBodyContent - '' ) '</jsp:body>' )

JspBodyBodyContent::= ScriptlessBody | Body | TagDependentBody
                     [ vc: JspBodyBodyContent ]

JspBodyEmptyBody   ::=     '/>'
                           |     '></jsp:body>'
                           |     <TRANSLATION_ERROR>
```

```
JspBodyParam      ::= S? '>' S? Param* '</jsp:body>'

JspBodyPluginTags ::= S? '>' S? PluginTags '</jsp:body>'

PluginTags        ::= ( '<jsp:params' Params S? )?
                      ( '<jsp:fallback' Fallback S? )?

Params            ::= '>' S?
                      (         (    '<jsp:body>'
                                     ( ( S? Param+ S? '</jsp:body>' )
                                       | <TRANSLATION_ERROR>
                                     )
                                )
                          |     Param+
                      )
                      '</jsp:params>'

Fallback          ::=       '/>'
                      |   (   '>' S? '<jsp:body>'
                                (         (    S?
                                               ( Char* - ( Char* '</jsp:body>' ) )
                                               '</jsp:body>' S?
                                          )
                                     |    <TRANSLATION_ERROR>
                                )
                                '</jsp:fallback>'
                          )
                      |   (   '>'
                                ( Char* - ( Char* '</jsp:fallback>' ) )
                                '</jsp:fallback>'
                          )

Param             ::= '<jsp:param' StdActionContent

Attributes        ::= ( S Attribute )* S?
                      [ vc: UniqueAttSpec ]

CustomAction      ::= TagPrefix ':' CustomActionName
                      [vc: CustomActionMatchesAndValid]

TagPrefix         ::= Name

CustomActionName  ::= Name
```

```
CustomActionBody  ::=     ( Attributes CustomActionEnd )
                      |   <TRANSLATION_ERROR>

CustomActionEnd   ::=     CustomActionTagDependent
                      |   CustomActionJSPContent
                      |   CustomActionScriptlessContent

CustomActionTagDependent::=  TagDependentOptionalBody
                             [vc: CustomActionTagDependentMatch]

CustomActionJSPContent::= OptionalBody
                          [ vc: CustomActionJSPContentMatch ]

CustomActionScriptlessContent::=  ScriptlessOptionalBody
                          [ vc: CustomActionScriptlessContentMatch ]

TemplateText      ::=     ( '<' | '${' )
                      |   ( TemplateChar* - ( TemplateChar* ( '<' | '${' ) ) )

TemplateChar      ::=     '\$'
                      |   '<\%'
                      |   Char
                      [ vc : QuotedDollarMatched ]

XMLTemplateText   ::=     ( S? '/>' )
                      |   (   S? '>'
                              (   ( Char* - ( Char* ( '<' | '${' ) ) )
                                  ( '${' ELExpressionBody )?
                                  CDSect?
                              )* ETag
                          )
                      |   <TRANSLATION_ERROR>
                      [ vc: ELEnabled ]

ExtraClosingTag   ::= ETag
                      [ vc: ExtraClosingTagMatch ]

ETag              ::= '</' TagPrefix ':' Name S? '>'
                      [ vc: ETagMatch ]
```

```
Attribute              ::=  Name Eq
                            (       ( '"<%=' RTAttributeValueDouble  )
                                |   ( "'<%=" RTAttributeValueSingle  )
                                |   ( '"'    AttributeValueDouble    )
                                |   ( "'"    AttributeValueSingle    )
                            )

NonRTAttribute         ::=  Name Eq
                            (       ( '"'    AttributeValueDouble    )
                                |   ( "'"    AttributeValueSingle    )
                            )

AnyAttributeValue      ::=  AttributeValue | RTAttributeValue

AttributeValue         ::=  AttributeValueDouble | AttributeValueSingle

RTAttributeValue       ::=  RTAttributeValueDouble | RTAttributeValueSingle

AttributeValueDouble::=  ( QuotedChar - '"' )*
                         ( '"' | <TRANSLATION_ERROR> )

AttributeValueSingle ::=  ( QuotedChar - "'" )*
                          ( "'" | <TRANSLATION_ERROR> )

RTAttributeValueDouble::=    ( ( QuotedChar - '"' )* -
                                   ( ( QuotedChar - '"' )* '%>' )
                               )
                               ( '%>"' | <TRANSLATION_ERROR> )
                               [ vc: RTAttributeScriptingEnabled ]
                               [ vc: ExpressionBodyContent ]

RTAttributeValueSingle::=    ( ( QuotedChar - "'" )* -
                                   ( ( QuotedChar - "'" )* '%>' )
                               )
                               ( "%>'" | <TRANSLATION_ERROR> )
                               [ vc: RTAttributeScriptingEnabled ]
                               [ vc: ExpressionBodyContent ]

Name                   ::=  XML::Name

Char                   ::=  XML::Char
```

```
QuotedChar      ::=     '&apos;'
                    |   '&quot;'
                    |   '\\'
                    |   '\"'
                    |   "\'"
                    |   '\$'
                    |   ( '${' ELExpressionBody )
                    |   Char
                    [ vc: QuotedDollarMatched ]

S               ::= XML::S

Eq              ::= XML::Eq

CDSect          ::= XML::CDSect
```

JSP.1.3.10.2 Validity Constraints

The following validity constraints are referenced in the above grammar using the syntax [vc: ValidityConstraint], and must be followed:

- ScriptingEnabled - The ScriptlessBody production must be followed if scripting is disabled for this translation unit. See the scripting-invalid JSP Configuration element (Section JSP.3.3.3).
- ScriptlessBody - The AllBody production cannot be followed if one of our parent nodes in the parse tree is a ScriptlessBody production. That is, once we have followed the ScriptlessBody production, until that production is complete we cannot choose the AllBody production.
- ELEnabled - The token ${ is not followed if expressions are disabled for this translation unit. See the isELIgnored page and tag directive (Section JSP.1.10.1 and Section JSP.8.5.1 respectively) and the el-ignored JSP Configuration element (Section JSP.3.3.2).
- TagFileSpecificDirectives - The JSPDirectiveBody production must be followed if the root production is JSPPage (i.e. this is a JSP page). The TagDefDirectiveBody production must be followed if the root production is JSPTagDef (i.e. this is a tag file).
- TagFileSpecificXMLDirectives - The XMLJSPDirectiveBody production must be followed if the root production is JSPPage (i.e. this is a JSP page). The XMLTagDefDirectiveBody production must be followed if the root production is JSPTagDef (i.e. this is a tag file).

- PageDirectiveUniqueAttr - A translation error will result if there is more than one occurrence of any attribute defined by this directive in a given translation unit, and if the value of the attribute is different than the previous occurrence. No translation error results if the value is identical to the previous occurrence. In addition, the import and pageEncoding attributes are excluded from this constraint (see Section JSP.1.10.1).
- TagLibDirectiveUniquePrefix - A translation error will result if the prefix AttributeValue has already previously been encountered as a potential TagPrefix in this translation unit.
- TagDirectiveUniqueAttr - A translation error will result if the prefix of this tag directive is already defined in the current scope, and if that prefix is bound to a namespace other than that specified by the uri or tagdir attribute.
- UniqueAttributeName - A translation error will result if there are two or more attribute directives with the same value for the name attribute in the same translation unit. A translation error will result if there is a variable directive with a name-given attribute equal to the value of the name attribute of an attribute directive in the same translation unit.
- UniqueVariableName - A translation error must occur if more than one variable directive appears in the same translation unit with the same value for the name-given attribute or the same value for the name-from-attribute attribute. A translation error must occur if there is a variable directive with a name-given attribute equal to the value of the name attribute of an attribute directive in the same translation unit. A translation error must occur if there is a variable directive with a name-from-attribute attribute whose value is not equal to the name attribute of an attribute directive in the same translation unit that is also of type java.lang.String, that is required and that is not an rtexprvalue. A translation error must occur if the value of the alias attribute is equal to the value of a name-given attribute of a variable directive, or the value of the name attribute of an attribute directive in the same translation unit.
- TagFileSpecificActions - The invoke and doBody standard actions are only matched if the JSPTagDef production was followed (i.e. if this is a tag file instead of a JSP page).
- RTAttributeScriptingEnabled - If the RTAttributeValueDouble or RTAttributeValueSingle productions are visited during parsing and scripting is disabled for this page, a translation error must be produced. See the scripting-invalid JSP Configuration element (Section JSP.3.3.3).

- ExpressionBodyContent - A translation error will result if the body content minus the closing delimiter (%>, or </jsp:expression>, depending on how the expression started) does not represent a well-formed expression in the scripting language selected for the JSP page.
- StdActionAttributesValid - An attribute is considered "provided" for this standard action if either the Attribute production or the AttributeBody production is followed in the context of the enclosing StandardAction production. A translation error will result if any of the following conditions is true:
  - The set of attributes "provided" for this standard action does not match one of the valid attribute combinations specified in Table JSP.1-5.
  - The same attribute is "provided" more than once, as determined by the attribute name.
  - An attribute is "provided" using the AttributeBody production that does not accept a request-time expression value, as indicated by the = prefix in Table JSP.1-5.
- StdActionBodyMatch - The StdActionBody production will only be matched if the production listed for this standard action in the "Body Production" column in Table JSP.1-5 is followed.
- AttributeBodyMatch - The type of element being specified determines which production is followed (see Section JSP.5.10, "<jsp:attribute>"for details):
  - If a custom action that specifies an attribute of type JspFragment, ScriptlessBody must be followed.
  - If a standard or custom action that accepts a request-time expression value, AllJspBody must be followed.
  - If a standard or custom action that does not accept a request-time expression value, TemplateTextBody must be followed.
- JspBodyBodyContent - The ScriptlessBody production must be followed if the body content for this tag is scriptless. The Body production must be followed if the body content for this tag is JSP. The TagDependentBody production must be followed if the body content for this tag is tagdependent.
- UniqueAttSpec - A translation error will result if the same attribute name appears more than once.
- CustomActionMatchesAndValid - Following the rules in Section JSP.7.3 for determining the relevant set of tags and tag libraries, assume the following:
  - Let U be the URI indicated by the uri AttributeValue of the previously encoun-

tered TagLibDirectiveAttrList with prefix matching the TagPrefix for this potential custom action, or nil if no such TagLibDirectiveAttrList was encountered in this translation unit.

- If U is not nil, let L be the <taglib> element in the relevant TLD entry such that L.uri is equal to U.

Then:

- If, after being parsed, the CustomAction production is matched (not yet taking into account the following rules), TagPrefix is considered a potential TagPrefix in this translation unit for the purposes of the TagLibDirectiveUniquePrefix validity constraint.
- The CustomAction production will not be matched if U is nil or if the TagPrefix does not match the prefix AttributeValue of a TagLibDirectiveAttrList previously encountered in this translation unit.
- Otherwise, if the CustomAction production is matched, a translation error will result if there does not exist a <tag> element T in a relevant TLD such that L.T.name is equal to CustomActionName.

- CustomActionTagDependentMatch - Assume the definition of L from the CustomActionMatchesAndValid validity constraint above. The CustomActionTagDependent production is not matched if there does not exist a <tag> element T in a relevant TLD such that L.T.body-content contains the value tagdependent.
- CustomActionJSPContentMatch - Assume the definition of L from the CustomActionMatchesAndValid validity constraint above. The CustomActionJSPContent production is not matched if there exists a <tag> element T in a relevant TLD such that L.T.body-content does not contain the value JSP.
- CustomActionScriptlessContentMatch - Assume the definition of L from the CustomActionMatchesAndValid validity constraint above. The CustomActionScriptlessContent production is not matched if there does not exist a <tag> element T in a relevant TLD such that L.T.body-content contains the value scriptless.
- QuotedDollarMatch - The '\$' token is only matched if EL is enabled for this translation unit. See Section JSP.3.3.2, "Deactivating EL Evaluation".
- ETagMatch - Assume the definition of U from the CustomActionMatchesAndValid validity constraint. If TagPrefix is not 'jsp' and U is nil, the ETag production is not matched. Otherwise, the ETag production is matched and a translation error will result if the prefix and name of this closing tag does not match the prefix and name of the starting tag at the corresponding nesting lev-

el, or if there is no corresponding nesting level (i.e. too many closing tags). This is similar to the way XML is defined, except that template text that looks like a closing element with an unrecognized prefix is allowed in the body of a custom or standard action. In the following example, assuming ‘my’ is a valid prefix and ‘indent’ is a valid tag, the tag is considered template text, and no translation error is produced:

```
<my:indent level=”2”>
    </ul>
</my:indent>
```

The following example, however, would produce a translation error, assuming ‘my’ is a valid prefix and ‘indent’ is a valid tag, and regardless of whether ‘othertag’ is a valid tag or not.

```
<my:indent level=”2”>
    </my:othertag>
</my:indent>
```

- ExtraClosingTagMatch - The ExtraClosingTag production is not matched if encountered within two or more nested Body productions (e.g. if encountered inside the body of a standard or custom action).

JSP.1.3.10.3 Standard Action Attributes

Table JSP.1-5 specifies, for each standard action element, the bodies and the attribute combinations that are valid. The value in the “Body Production” column specifies a production name that must be matched for the body of the standard action to be considered valid. The value in the “Valid Attribute Combinations” column uses the same syntax as the attrset notation described at the start of Section JSP.1.3.10, and indicates which attributes can be provided. Note that for some valid attribute combinations, there are differing body productions. The first

attribute combination to be matched selects the valid body production for this standard action invocation.

Table JSP.1-5 *Valid body content and attributes for Standard Actions*

Element	Body Production	Valid Attribute Combinations
jsp:useBean	OptionalBody	(!id, scope, !class)
	OptionalBody	(!id, scope, !type)
	OptionalBody	(!id, scope, !class, !type)
	OptionalBody	(!id, scope, =!beanName, !type)
jsp:setProperty	EmptyBody	(!name, !property, param)
	EmptyBody	(!name, !property, =!value)
jsp:getProperty	EmptyBody	(!name, !property)
jsp:include	ParamBody	(=!page, flush)
jsp:forward	ParamBody	(=!page)
jsp:plugin	PluginBody	(!type, !code, !codebase, align, archive, =height, hspace, jreversion, name, vspace, title, =width, nspluginurl, iepluginurl, mayscript)
jsp:invoke	EmptyBody	(!fragment, !var, scope)
	EmptyBody	(!fragment, !varReader, scope)
	EmptyBody	(!fragment)
jsp:doBody	EmptyBody	(!var, scope)
	EmptyBody	(!varReader, scope)
	EmptyBody	()
jsp:element	OptionalBody	(=!name)
jsp:output	EmptyBody	(omit-xml-declaration)
	EmptyBody	(omit-xml-declaration, !doctype-root-element, !doctype-system, doctype-public)
jsp:param	EmptyBody	(!name, =!value)

JSP.1.4 Error Handling

Errors may occur at translation time or at request time. This section describes how errors are treated by a compliant implementation.

JSP.1.4.1 Translation Time Processing Errors

The translation of a JSP page source into a corresponding JSP page implementation class by a JSP container can occur at any time between initial deployment of the JSP page into the JSP container and the receipt and processing of a client request for the target JSP page. If translation occurs prior to the receipt of a client request for the target JSP page, error processing and notification is implementation dependent and not covered by this specification. In all cases, fatal translation failures shall result in the failure of subsequent client requests for the translation target with the appropriate error specification: For HTTP protocols the error status code 500 (Server Error) is returned.

JSP.1.4.2 Request Time Processing Errors

During the processing of client requests, errors can occur in either the body of the JSP page implementation class, or in some other code (Java or other implementation programming language) called from the body of the JSP page implementation class. Runtime errors occurring are realized in the page implementation, using the Java programming language exception mechanism to signal their occurrence to caller(s) of the offending behavior[1].

These exceptions may be caught and handled (as appropriate) in the body of the JSP page implementation class.

Any uncaught exceptions thrown in the body of the JSP page implementation class result in the forwarding of the client request and uncaught exception to the errorPage URL specified by the JSP page (or the implementation default behavior, if none is specified).

Information about the error is passed as javax.servlet.ServletRequest attributes to the error handler, with the same attributes as specified by the Servlet specification. Names starting with the prefixes java and javax are reserved by the

---

[1] Note that this is independent of scripting language. This specification requires that unhandled errors occurring in a scripting language environment used in a JSP container implementation to be signalled to the JSP page implementation class via the Java programming language exception mechanism.

different specifications of the Java platform. The javax.servlet prefix is reserved and used by the servlet and JSP specifications.

JSP.1.4.3 Using JSPs as Error Pages

A JSP is considered an Error Page if it sets the page directive's isErrorPage attribute to true. If a page has isErrorPage set to true, then the "exception" implicit scripting language variable (see Table JSP.1-7) of that page is initialized. The variable is set to the value of the javax.servlet.error.exception request attribute value if present, otherwise to the value of the javax.servlet.jsp.jspException request attribute value (for backwards compatibility for JSP pages pre-compiled with a JSP 1.2 compiler).

In addition, an ErrorData instance must be initialized based on the error handler ServletRequest attributes defined by the Servlet specification, and made available through the PageContext to the page. This has the effect of providing easy access to the error information via the Expression Language. For example, an Error Page can access the status code using the syntax ${pageContext.errorData.statusCode}. See Chapter JSP.12 for details.

JSP.1.5 Comments

There are two types of comments in a JSP page: comments to the JSP page itself, documenting what the page is doing; and comments that are intended to appear in the generated document sent to the client.

JSP.1.5.1 Generating Comments in Output to Client

In order to generate comments that appear in the response output stream to the requesting client, the HTML and XML comment syntax is used, as follows:

<!-- *comments ...* -->

These comments are treated as uninterpreted template text by the JSP container. Dynamic content that appears within HTML/XML comments, such as actions, scriptlets and expressions, is still processed by the container. If the generated comment is to have dynamic data, this can be obtained through an expression syntax, as in:

<!-- *comments* <%= expression %> *more comments ...* -->

JSP.1.5.2 JSP Comments

A JSP comment is of the form

```
<%-- anything but a closing --%> ... --%>
```

The body of the content is ignored completely. Comments are useful for documentation but also are used to “comment out” some portions of a JSP page. Note that JSP comments do not nest.

An alternative way to place a comment in JSP is to use the comment mechanism of the scripting language. For example:

```
<% /** this is a comment ... **/ %>
```

JSP.1.6 Quoting and Escape Conventions

The following quoting conventions apply to JSP pages.

Note – The current quoting rules do not allow for quoting special characters such as \n - the only current way to do this in a JSP is with a Java expression.

Quoting in EL Expressions

- There is no special quoting mechanism within EL expressions; use a literal ‘${‘ if the literal ${ is desired and expressions are enabled for the page. For example, the evaluation of ${‘${‘} is ‘${‘. Note that ${‘}’} is legal, and simply evaluates to ‘}’.

Quoting in Scripting Elements

- A literal %> is quoted by %\>

Quoting in Template Text

- A literal <% is quoted by <\%
- Only when the EL is enabled for a page (see Section JSP.3.3.2, “Deactivating EL Evaluation”), a literal $ can be quoted by \$. This is not required but is useful for quoting EL expressions.

Quoting in Attributes

Quotation is done consistently regardless of whether the attribute value is a literal or a request-time attribute expression. Quoting can be used in attribute values regardless of whether they are delimited using single or double quotes. It is only required as described below.

- A ‘ is quoted as \'. This is required within a single quote-delimited attribute value.
- A “ is quoted as \". This is required within a double quote-delimited attribute value.
- A \ is quoted as \\
- Only when the EL is enabled for a page (see Section JSP.3.3.2, “Deactivating EL Evaluation”), a literal $ can be quoted by \$. This is not required but is useful for quoting EL expressions.
- A %> is quoted as %\>
- A <% is quoted as <\%
- The entities ' and " are available to describe single and double quotes.

Examples

The following line shows an illegal attribute values.

```
<mytags:tag value="<%= "hi!" %>" />
```

The following line shows a legal scriptlet, but perhaps with an intended value. The result is Joe said %\> not Joe said %>.

```
<%= "Joe said %\\>" %>
```

The next lines are all legal quotations.

```
<%= "Joe said %/>" %>
```

```
<%= "Joe said %\>" %>
```

```
<% String joes_statement = "hi!"; %>
                    <%= "Joe said \"" + joes_statement + "\"." %>
                    <x:tag value='<%="Joe said \\"" + joes_statement + "\\"."%>'/>
```

```
<x:tag value='<%= "hi!" %>' />

<x:tag value="<%= \"hi!\" %>" />

<x:tag value='<%= \"name\" %>' />

<x:tag value="<%= \"Joe said 'hello'\" %>"/>

<x:tag value="<%= \"Joe said \\\"hello\\\" \" %>"/>

<x:tag value="end expression %\>"/>

<% String s="abc"; %>
                    <x:tag value="<%= s + \"def\" + \"jkl\" + 'm' + \'n\' %>" />
                    <x:tag value='<%= s + \"def\" + "jkl" + \'m\' + \'n\' %>' />
```

XML Documents

The quoting conventions are different from those of XML. See Chapter JSP.6.

JSP.1.7 Overall Semantics of a JSP Page

A JSP page implementation class defines a _jspService() method mapping from the *request* to the *response* object. Some details of this transformation are specific to the scripting language used (see Chapter JSP.9). Most details are not language specific and are described in this chapter.

The content of a JSP page is devoted largely to describing the data that is written into the output stream of the response. (The JSP container usually sends this data back to the client.) The description is based on a JspWriter object that is exposed through the implicit object out (see Section JSP.1.8.3, “Implicit Objects”). Its value varies:

- Initially, out is a new JspWriter object. This object may be different from the stream object returned from response.getWriter(), and may be considered to be interposed on the latter in order to implement buffering (see Section JSP.1.10.1, “The page Directive”). This is the initial out object. JSP page authors are prohibited from writing directly to either the PrintWriter or OutputStream associated with the ServletResponse.
- The JSP container should not invoke response.getWriter() until the time when the first portion of the content is to be sent to the client. This enables a number of uses of JSP, including using JSP as a language to “glue” actions that deliver

binary content, or reliably forwarding to a servlet, or change dynamically the content type of the respose before generating content. See Chapter JSP.4.

- Within the body of some actions, out may be temporarily re-assigned to a different (nested) instance of a JspWriter object. Whether this is the case depends on the details of the action's semantics. Typically the content of these temporary streams is appended to the stream previously referred to by out, and out is subsequently re-assigned to refer to the previous (nesting) stream. Such nested streams are always buffered, and require explicit flushing to a nesting stream or their contents will be discarded.
- If the initial out JspWriter object is buffered, then depending upon the value of the autoFlush attribute of the page directive, the content of that buffer will either be automatically flushed out to the ServletResponse output stream to obviate overflow, or an exception shall be thrown to signal buffer overflow. If the initial out JspWriter is unbuffered, then content written to it will be passed directly through to the ServletResponse output stream.

A JSP page can also describe what should happen when some specific events occur. In JSP 2.0, the only events that can be described are the initialization and the destruction of the page. These events are described using "well-known method names" in declaration elements. (See Section JSP.11.1.1.1).

JSP.1.8 Objects

A JSP page can access, create, and modify server-side objects. Objects can be made visible to actions, EL expressions and to scripting elements. An object has a *scope* describing what entities can access the object.

Actions can access objects using a name in the PageContext object.

An object exposed through a scripting variable has a scope within the page. Scripting elements can access some objects directly via a *scripting variable*. Some implicit objects are visible via scripting variables and EL expressions in any JSP page.

JSP.1.8.1 Objects and Variables

An object may be made accessible to code in the scripting elements through a scripting language variable. An element can define scripting variables that will contain, at process request-time, a reference to the object defined by the element, although other references may exist depending on the scope of the object.

An element type indicates the name and type of such variables although details on the name of the variable may depend on the Scripting Language. The scripting language may also affect how different features of the object are exposed. For example, in the JavaBeans specification, properties are exposed via getter and setter methods, while these properties are available directly as variables in the JavaScript™ programming language.

The exact rules for the visibility of the variables are scripting language specific. Chapter JSP.1.1 defines the rules for when the language attribute of the page directive is java.

JSP.1.8.2 Objects and Scopes

A JSP page can create and/or access some Java objects when processing a request. The JSP specification indicates that some objects are created implicitly, perhaps as a result of a directive (see Section JSP.1.8.3, “Implicit Objects”). Other objects are created explicitly through actions, or created directly using scripting code. Created objects have a scope attribute defining where there is a reference to the object and when that reference is removed.

The created objects may also be visible directly to scripting elements through scripting-level variables (see Section JSP.1.8.3, “Implicit Objects”).

Each action and declaration defines, as part of its semantics, what objects it creates, with what scope attribute, and whether they are available to the scripting elements.

Objects are created within a JSP page instance that is responding to a request object. There are several scopes:

- page - Objects with page scope are accessible only within the page where they are created. All references to such an object shall be released after the response is sent back to the client from the JSP page or the request is forwarded somewhere else. References to objects with page scope are stored in the pageContext object.
- request - Objects with request scope are accessible from pages processing the same request where they were created. References to the object shall be released after the request is processed. In particular, if the request is forwarded to a resource in the same runtime, the object is still reachable. References to objects with request scope are stored in the request object.
- session - Objects with session scope are accessible from pages processing requests that are in the same session as the one in which they were created. It is not legal to define an object with session scope from within a page that is not

session-aware (see Section JSP.1.10.1, “The page Directive”). All references to the object shall be released after the associated session ends. References to objects with session scope are stored in the session object associated with the page activation.

- application - Objects with application scope are accessible from pages processing requests that are in the same application as they one in which they were created. Objects with application scope can be defined (and reached) from pages that are not session-aware. References to objects with application scope are stored in the application object associated with a page activation. The application object is the servlet context obtained from the servlet configuration object. All references to the object shall be released when the runtime environment reclaims the ServletContext.

A name should refer to a unique object at all points in the execution; that is, all the different scopes really should behave as a single name space. A JSP container implementation may or may not enforce this rule explicitly for performance reasons.

JSP.1.8.3 Implicit Objects

JSP page authors have access to certain implicit objects that are always available for use within scriptlets and scriptlet expressions through scripting variables that are declared implicitly at the beginning of the page. All scripting languages are required to provide access to these objects. See Section JSP.2.2.3 for the implicit objects available within EL expressions. Implicit objects are available to tag handlers through the pageContext object, see below.

Each implicit object has a class or interface type defined in a core Java technology or Java Servlet API package, as shown in **Table JSP.1-6**.

Table JSP.1-6 *Implicit Objects Available in JSP Pages*

Variable Name	Type	Semantics & Scope
request	protocol dependent subtype of: javax.servlet.ServletRequest e.g: javax.servlet.http.HttpServletRequest	The request triggering the service invocation. request scope.

Table JSP.1-6 *Implicit Objects Available in JSP Pages*

Variable Name	Type	Semantics & Scope
response	protocol dependent subtype of: javax.servlet.ServletResponse, e.g: javax.servlet.http.HttpServletResponse	The response to the request. page scope.
pageContext	javax.servlet.jsp.PageContext	The page context for this JSP page. page scope.
session	javax.servlet.http.HttpSession	The session object created for the requesting client (if any). This variable is only valid for HTTP protocols. session scope
application	javax.servlet.ServletContext	The servlet context obtained from the servlet configuration object (as in the call getServlet-Config(). getContext()) application scope
out	javax.servlet.jsp.JspWriter	An object that writes into the output stream. page scope
config	javax.servlet.ServletConfig	The ServletConfig for this JSP page page scope
page	java.lang.Object	The instance of this page's implementation class processing the current request[a] page scope

a. When the scripting language is java then page is a synonym for this in the body of the page.

In addition, the exception implicit object can be accessed in an error page, as described in Table JSP.1-7.

Table JSP.1-7 *Implicit Objects Available in Error Pages*

Variable Name	Type	Semantics & Scope
exception	java.lang.Throwable	The uncaught Throwable that resulted in the error page being invoked. page scope.

Object names with prefixes jsp, _jsp, jspx and _jspx, in any combination of upper and lower case, are reserved by the JSP specification.

See Section JSP.7.5.1 for some non-normative conventions for the introduction of new implicit objects.

JSP.1.8.4 The pageContext Object

A PageContext is an object that provides a context to store references to objects used by the page, encapsulates implementation-dependent features, and provides convenience methods. A JSP page implementation class can use a PageContext to run unmodified in any compliant JSP container while taking advantage of implementation-specific improvements like high performance JspWriters.

See Chapter JSP.12 for more details.

JSP.1.9 Template Text Semantics

The semantics of *template (or uninterpreted) Text* is very simple: the template text is passed through to the current out JspWriter implicit object, after applying the substitutions of Section JSP.1.6, “Quoting and Escape Conventions”.

JSP.1.10 Directives

Directives are messages to the JSP container. Directives have this syntax:

```
<%@ directive { attr=”value” }* %>
```

There may be optional white space after the <%@ and before %>.

This syntax is easy to type and concise but it is not XML-compatible. Chapter JSP.6 describes equivalent alternative mechanisms that are consistent with XML syntax.

Directives do not produce any output into the current out stream.

There are three directives: the page and the taglib directives are described next, while the include directive is described in “The include Directive” on page 51.

JSP.1.10.1 The page Directive

The page directive defines a number of page dependent properties and communicates these to the JSP container.

This <jsp:directive.page> element (Section JSP.6.3.4) describes the same information following the XML syntax.

A translation unit (JSP source file and any files included via the include directive) can contain more than one instance of the page directive, all the attributes will apply to the complete translation unit (i.e. page directives are position independent). An exception to this position independence is the use of the pageEncoding and contentType attributes in the determination of the page character encoding; for this purpose, they should appear at the beginning of the page (see Section JSP.4.1). There shall be only one occurrence of any attribute/value pair defined by this directive in a given translation unit, unless the values for the duplicate attributes are identical for all occurrences. The import and pageEncoding attributes are exempt from this rule and can appear multiple times. Multiple uses of the import attribute are cumulative (with ordered set union semantics). The pageEncoding attribute can occur at most once per file (or a translation error will result), and applies only to the file in which it appears. Other such multiple attribute/value (re)definitions result in a fatal translation error if the values do not match.

The attribute/value namespace is reserved for use by this, and subsequent, JSP specification(s).

Unrecognized attributes or values result in fatal translation errors.

Examples

The following directive provides some user-visible information on this JSP page:

```
<%@ page info=”my latest JSP Example” %>
```

The following directive requests no buffering, and provides an error page.

```
<%@ page buffer="none" errorPage="/oops.jsp" %>
```

The following directive indicates that the scripting language is based on Java, that the types declared in the package com.myco are directly available to the scripting code, and that a buffering of 16KB should be used.

```
<%@ page language="java" import="com.myco.*" buffer="16kb" %>
```

Syntax

```
<%@ page page_directive_attr_list %>

page_directive_attr_list ::= { language="scriptingLanguage"}
                             { extends="className"          }
                             { import="importList"          }
                             { session="true|false"         }
                             { buffer="none|sizekb"         }
                             { autoFlush="true|false"       }
                             { isThreadSafe="true|false"    }
                             { info="info_text"             }
                             { errorPage="error_url"        }
                             { isErrorPage="true|false"     }
                             { contentType="ctinfo"         }
                             { pageEncoding="peinfo"        }
                             { isELIgnored="true|false"     }
```

The details of the attributes are as follows:

Table JSP.1-8 *Page Directive Attributes*

language	Defines the scripting language to be used in the scriptlets, expression scriptlets, and declarations within the body of the translation unit (the JSP page and any files included using the include directive below). In JSP 2.0, the only defined and required scripting language value for this attribute is java. This specification only describes the semantics of scripts for when the value of the language attribute is java. When java is the value of the scripting language, the Java Programming Language source code fragments used within the translation unit are required to conform to the Java Programming Language Specification in the way indicated in Chapter JSP.9. All scripting languages must provide some implicit objects that a JSP page author can use in declarations, scriptlets, and expressions. The specific objects that can be used are defined in Section JSP.1.8.3, “Implicit Objects”.” All scripting languages must support the Java Runtime Environment (JRE). All scripting languages must expose the Java technology object model to the script environment, especially implicit variables, JavaBeans component properties, and public methods. Future versions of the JSP specification may define additional values for the language attribute and all such values are reserved. It is a fatal translation error for a directive with a non-java language attribute to appear after the first scripting element has been encountered. Default is java.
extends	The value is a fully qualified Java programming language class name, that names the superclass of the class to which this JSP page is transformed (see Chapter JSP.11). This attribute should not be used without careful consideration as it restricts the ability of the JSP container to provide specialized superclasses that may improve on the quality of rendered service. See Section JSP.7.5.1 for an alternate way to introduce objects into a JSP page that does not have this drawback.

Table JSP.1-8 *Page Directive Attributes*

import	An import attribute describes the types that are available to the scripting environment. The value is as in an import declaration in the Java programming language, a (comma separated) list of either a fully qualified Java programming language type name denoting that type, or of a package name followed by the .* string, denoting all the public types declared in that package. The import list shall be imported by the translated JSP page implementation and is thus available to the scripting environment. The default import list is java.lang.*, javax.servlet.*, javax.servlet.jsp.* and javax.servlet.http.*. This attribute is currently only defined when the value of the language directive is java.
session	Indicates that the page requires participation in an (HTTP) session. If true then the implicit script language variable named session of type javax.servlet.http.HttpSession references the current/new session for the page. If false then the page does not participate in a session; the session implicit variable is unavailable, and any reference to it within the body of the JSP page is illegal and shall result in a fatal translation error. Default is true.
buffer	Specifies the buffering model for the initial out JspWriter to handle content output from the page. If none, then there is no buffering and all output is written directly through to the ServletResponse PrintWriter. The size can only be specified in kilobytes. The suffix kb is mandatory or a translation error must occur. If a buffer size is specified then output is buffered with a buffer size not less than that specified. Depending upon the value of the autoFlush attribute, the contents of this buffer is either automatically flushed, or an exception is raised, when overflow would occur. The default is buffered with an implementation buffer size of not less than 8kb.

Table JSP.1-8 *Page Directive Attributes*

autoFlush	Specifies whether the buffered output should be flushed automatically (true value) when the buffer is filled, or whether an exception should be raised (false value) to indicate buffer overflow. It is illegal, resulting in a translation error, to set autoFlush to false when buffer=none. The default value is true.
isThreadSafe	***Note: The Servlet 2.4 specification deprecates SingleThreadModel, which is the most common mechanism for JSP containers to implement isThreadSafe. Page authors are advised against using isThreadSafe, as the generated Servlet may contain deprecated code.*** Indicates the level of thread safety implemented in the page. If false then the JSP container shall dispatch multiple outstanding client requests, one at a time, in the order they were received, to the page implementation for processing. If true then the JSP container may choose to dispatch multiple outstanding client requests to the page simultaneously. Page authors using true must ensure that they properly synchronize access to the shared state of the page. Default is true. Note that even if the isThreadSafe attribute is false the JSP page author must ensure that accesses to any shared objects are properly synchronized., The objects may be shared in either the ServletContext or the HttpSession .
info	Defines an arbitrary string that is incorporated into the translated page, that can subsequently be obtained from the page's implementation of Servlet.getServletInfo method.
isErrorPage	Indicates if the current JSP page is intended to be the URL target of another JSP page's errorPage. If true, then the implicit script language variable exception is defined and its value is a reference to the offending Throwable from the source JSP page in error. If false then the exception implicit variable is unavailable, and any reference to it within the body of the JSP page is illegal and shall result in a fatal translation error. Default is false.

Table JSP.1-8 *Page Directive Attributes*

errorPage	Defines a URL to a resource to which any Java programming language Throwable object(s) thrown but not caught by the page implementation are forwarded for error processing. The provided URL spec is as in Section JSP.1.2.1. If the URL names another JSP page then, when invoked that JSP page's exception implicit script variable shall contain a reference to the originating uncaught Throwable. The default URL is implementation dependent. Note the Throwable object is transferred by the throwing page implementation to the error page implementation by saving the object reference on the common ServletRequest object using the setAttribute method, with a name of javax.servlet.jsp.jspException (for backwards-compatibility) and also javax.servlet.error.exception (for compatibility with the servlet specification). See Section JSP.1.4.3 for more details). Note: if autoFlush=true then if the contents of the initial JspWriter has been flushed to the ServletResponse output stream then any subsequent attempt to dispatch an uncaught exception from the offending page to an errorPage may fail. If the page defines an error page via the page directive, any error pages defined in web.xml will not be used.
contentType	Defines the MIME type and the character encoding for the response of the JSP page, and is also used in determining the character encoding of the JSP page. Values are either of the form "TYPE" or "TYPE;charset=CHARSET"with an optional white space after the ";". "TYPE" is a MIME type, see the IANA registry at http://www.iana.org/assignments/media-types/index.html for useful values. "CHARSET", if present, must be the IANA name for a character encoding. The default value for "TYPE" is "text/html" for JSP pages in standard syntax, or "text/xml" for JSP documents in XML syntax. If "CHARSET" is not specified, the response character encoding is determined as described in Section JSP.4.2, "Response Character Encoding". See Chapter JSP.4 for complete details on character encodings.

Table JSP.1-8 *Page Directive Attributes*

pageEncoding	Describes the character encoding for the JSP page. The value is of the form "CHARSET", which must be the IANA name for a character encoding. For JSP pages in standard syntax, the character encoding for the JSP page is the charset given by the pageEncoding attriute if it is present, otherwise the charset given by the contentType attribute if it is present, otherwise "ISO-8859-1". For JSP documents in XML syntax, the character encoding for the JSP page is determined as described in section 4.3.3 and appendix F.1 of the XML specification. The pageEncoding attribute is not needed for such documents. It is a translation-time error if a document names different encodings in its XML prolog / text declaration and in the pageEncoding attribute. The corresponding JSP configuration element is page-encoding (see Section JSP.3.3.4, "Declaring Page Encodings"). See Chapter JSP.4 for complete details on character encodings.
isELIgnored	Defines whether EL expressions are ignored or evaluated for this page and translation unit. If true, EL expressions (of the form ${...}) are ignored by the container. If false, EL expressions (of the form ${...}) are evaluated when they appear in template text or action attributes. The corresponding JSP configuration element is el-ignored (see Section JSP.3.3.2). The default value varies depending on the web.xml version - see Section JSP.2.2.4, "Deactivating EL Evaluation".

JSP.1.10.2 The taglib Directive

The set of significant tags a JSP container interprets can be extended through a tag library.

The taglib directive in a JSP page declares that the page uses a tag library, uniquely identifies the tag library using a URI and associates a tag prefix that will distinguish usage of the actions in the library.

If a JSP container implementation cannot locate a tag library description, a fatal translation error shall result.

It is a fatal translation error for the taglib directive to appear after actions or functions using the prefix.

A tag library may include a validation method that will be consulted to determine if a JSP page is correctly using the tag library functionality.

See Chapter JSP.7 for more specification details. And see Section JSP.7.2.3 for an implementation note.

Section JSP.6.3.1 describes how the functionality of this directive can be exposed using XML syntax.

Examples

In the following example, a tag library is introduced and made available to this page using the super prefix; no other tag libraries should be introduced in this page using this prefix. In this particular case, we assume the tag library includes a doMagic element type, which is used within the page.

```
<%@ taglib uri="http://www.mycorp/supertags" prefix="super" %>

...

<super:doMagic>

...

</super:doMagic>
```

Syntax

```
<%@ taglib ( uri="tagLibraryURI" | tagdir="tagDir" ) prefix="tagPrefix" %>
```

where the attributes are:

Table JSP.1-9

uri	Either an absolute URI or a relative URI specification that uniquely identifies the tag library descriptor associated with this prefix. The URI is used to locate a description of the tag library as indicated in Chapter 7.

Table JSP.1-9

tagdir	Indicates this prefix is to be used to identify tag extensions installed in the /WEB-INF/tags/ directory or a subdirectory. An implicit tag library descriptor is used (see Section JSP.8.4 for details). A translation error must occur if the value does not start with /WEB-INF/tags/. A translation error must occur if the value does not point to a directory that exists. A translation error must occur if used in conjunction with the uri attribute.
prefix	Defines the prefix string in <prefix>:<tagname> that is used to distinguish a custom action, e.g <myPrefix:myTag>. Prefixes starting with jsp:, jspx:, java:, javax:, servlet:, sun:, and sunw: are reserved. A prefix must follow the naming convention specified in the XML namespaces specification. Empty prefixes are illegal in this version of the specification, and must result in a translation error.

A fatal translation-time error will result if the JSP page translator encounters a tag with name prefix: Name using a prefix that is introduced using the taglib directive, and Name is not recognized by the corresponding tag library.

JSP.1.10.3 The include Directive

The include directive is used to substitute text and/or code at JSP page translation-time. The <%@ include file="relativeURLspec" %> directive inserts the text of the specified resource into the page or tag file. The included file is subject to the access control available to the JSP container. The file attribute is as in Section JSP.1.2.1.

With respect to the standard and XML syntaxes, a file included via the include directive can use either the same syntax as the including page, or a different syntax. the semantics for mixed syntax includes are described in Section JSP.1.10.5.

A JSP container can include a mechanism for being notified if an included file changes, so the container can recompile the JSP page. However, the JSP 2.0 specification does not have a way of directing the JSP container that included files have changed.

The <jsp:directive.include> element (Section JSP.6.3.5) describes the same information following the XML syntax.

Examples

The following example requests the inclusion, at translation time, of a copyright file. The file may have elements which will be processed too.

```
<%@ include file=”copyright.html” %>
```

Syntax

```
<%@ include file="relativeURLspec" %>
```

JSP.1.10.4 Implicit Includes

Many JSP pages start with a list of taglib directives that activate the use of tag libraries within the page. In some cases, these are the only tag libraries that are supposed to be used by the JSP page authors. These, and other common conventions are greately facilitated by two JSP configuration elements: include-prelude and include-coda. A full description of the mechanism is in Section JSP.3.3.5.

With respect to the standard and XML syntaxes, just as with the include directive, implicit includes can use either the same syntax as the including page, or a different syntax. The semantics for mixed syntax includes are described in Section JSP.1.10.5.

JSP.1.10.5 Including Data in JSP Pages

Including data is a significant part of the tasks in a JSP page. Accordingly, the JSP 2.0 specification has two include mechanisms suited to different tasks. A summary of their semantics is shown in Table JSP.1-10.

Table JSP.1-10 *Summary of Include Mechanisms in JSP 2.0*

Syntax	Spec	Object	Description	Section
Include Directive - Translation-time				
<%@ include file=... %>	file-relative	static	Content is parsed by JSP container.	JSP.1.10.3
Include Action - Request-time				
<jsp:include page= />	page-relative	static and dynamic	Content is not parsed; it is included in place.	JSP.5.4

The Spec column describes what type of specification is valid to appear in the given element. The JSP specification requires a relative URL spec. The reference is resolved by the web/application server and its URL map is involved. Include directives are interpreted relative to the current JSP file; jsp:include actions are interpreted relative to the current JSP page.

An include directive regards a resource like a JSP page as a static object; i.e. the text in the JSP page is included. An include action regards a resource like a JSP page as a dynamic object; i.e. the request is sent to that object and the result of processing it is included.

Implicit include directives can also be requested for a collection of pages through the use of the <include-prelude> and <include-coda> elements of the JSP configuration section of web.xml.

For translation-time includes, included content can use either the same syntax as the including page, or a different syntax. For example, a JSP file written in the standard JSP syntax can include a JSP file written using the XML syntax. The following semantics for translation-time includes apply:

- The JSP container must detect the syntax for each JSP file individually and parse each JSP file according to the syntax in which it is written.
- A JSP file written using the XML syntax must be well-formed according to the "XML" and "Namespaces in XML" specifications, otherwise a translation error must occur.
- When including a JSP document (written in the XML syntax), in the resulting XML View of the translation unit the root element of the included segment must have the default namespace reset to "". This is so that any namespaces associated with the empty prefix in the including document are not carried over to the included document.
- When a taglib directive is encountered in a standard syntax page, the namespace is applied globally, and is added to the <jsp:root> element of the resulting XML View of the translation unit.
- If a taglib directive is encountered in a standard syntax page that attempts to redefine a prefix that is already defined in the current scope (by a JSP segment in either syntax), a translation error must occur unless that prefix is being re-defined to the same namespace URI.

See Section JSP.10.3 for examples of how these semantics are applied to actual JSP pages and documents.

JSP.1.10.6 Additional Directives for Tag Files

Additional directives are available when editing a tag file. See Section JSP.8.5, “Tag File Directives” for details.

JSP.1.11 EL Elements

EL expressions can appear in template data and in attribute values. EL expressions a defined in more detail in Chapter 2.

EL expressions can be disabled through the use of JSP configuration elements and page directives; see Section JSP.1.10.1 and Section JSP.3.3.2.

EL expressions, when not disabled, can be used anywhere within template data.

EL expressions can be used in any attribute of a standard action that this specification indicates can accept a run-time expression value, and in any attribute of a custom action that has been indicated to accept run-time expressions (i.e. their associated <rtexprvalue> in the TLD is true; see Appendix JSP.C).

JSP.1.12 Scripting Elements

Scripting elements are commonly used to manipulate objects and to perform computation that affects the content generated.

JSP 2.0 adds EL expressions as an alternative to scripting elements. These are described in more detail in Chapter JSP.2. Note that scripting elements can be disabled through the use of the scripting-invalid element in the web.xml deployment descriptor (see Section JSP.3.3.3).

There are three other classes of scripting elements: *declarations*, *scriptlets* and *expressions*. The scripting language used in the current page is given by the value of the language directive (see Section JSP.1.10.1, “The page Directive”). In JSP 2.0, the only value defined is java.

Declarations are used to declare scripting language constructs that are available to all other scripting elements. Scriptlets are used to describe actions to be performed in response to some request. Scriptlets that are program fragments can also be used to do things like iterations and conditional execution of other elements in the JSP page. Expressions are complete expressions in the scripting language that get evaluated at response time; commonly, the result is converted into a string and inserted into the output stream.

All JSP containers must support scripting elements based on the Java programming language. Additionally, JSP containers may also support other scripting languages. All such scripting languages must support:

- Manipulation of Java objects.
- Invocation of methods on Java objects.
- Catching of Java language exceptions.

The precise definition of the semantics for scripting done using elements based on the Java programming language is given in Chapter JSP.9.

The semantics for other scripting languages are not precisely defined in this version of the specification, which means that portability across implementations cannot be guaranteed. Precise definitions may be given for other languages in the future.

Each scripting element has a <%-based syntax as follows:

```
<%! this is a declaration %>
<% this is a scriptlet %>
<%= this is an expression %>
```

White space is optional after <%!, <%, and <%=, and before %>.

The equivalent XML elements for these scripting elements are described in Section JSP.6.3.7.

JSP.1.12.1 Declarations

Declarations are used to declare variables and methods in the scripting language used in a JSP page. A declaration must be a complete declarative statement, or sequence thereof, according to the syntax of the scripting language specified.

Declarations do not produce any output into the current out stream.

Declarations are initialized when the JSP page is initialized and are made available to other declarations, scriptlets, and expressions.

The <jsp:declaration> element (Section JSP.6.3.7) describes the same information following the XML syntax.

Examples

For example, the first declaration below declares an integer, global to the page. The second declaration does the same and initializes it to zero. This type of initialization should be done with care in the presence of multiple requests

on the page. The third declaration declares a method global to the page.

```
<%! int i; %>

<%! int i = 0; %>

<%! public String f(int i) { if (i<3) return("..."); ... } %>
```

Syntax

```
<%! declaration(s) %>
```

JSP.1.12.2 Scriptlets

Scriptlets can contain any code fragments that are valid for the scripting language specified in the language attribute of the page directive. Whether the code fragment is legal depends on the details of the scripting language (see Chapter JSP.9).

Scriptlets are executed at request-processing time. Whether or not they produce any output into the out stream depends on the code in the scriptlet. Scriptlets can have side-effects, modifying the objects visible to them.

When all scriptlet fragments in a given translation unit are combined in the order they appear in the JSP page, they must yield a valid statement, or sequence of statements, in the specified scripting language.

To use the %> character sequence as literal characters in a scriptlet, rather than to end the scriptlet, escape them by typing %\>.

The <jsp:scriptlet> element (Section JSP.6.3.7) describes the same information following the XML syntax.

Examples

Here is a simple example where the page changed dynamically depending on the time of day.

```
<% if (Calendar.getInstance().get(Calendar.AM_PM) == Calendar.AM) {%>
Good Morning
<% } else { %>
Good Afternoon
<% } %>
```

A scriptlet can also have a local variable declaration, for example the following scriptlet just declares and initializes an integer, and later increments it.

```
<% int i; i= 0; %>
About to increment i...
<% i++ %>
```

Syntax

```
<% scriptlet %>
```

JSP.1.12.3 Expressions

An expression element in a JSP page is a scripting language expression that is evaluated and the result is coerced to a String. The result is subsequently emitted into the current out JspWriter object.

If the result of the expression cannot be coerced to a String the following must happen: If the problem is detected at translation time, a translation time error shall occur. If the coercion cannot be detected during translation, a ClassCastException shall be raised at request time.

A scripting language may support side-effects in expressions when the expression is evaluated. Expressions are evaluated left-to-right in the JSP page. If an expression appears in more than one run-time attribute, they are evaluated left-to-right in the tag. An expression might change the value of the out object, although this is not something to be done lightly.

The expression must be a complete expression in the scripting language in which it is written, or a translation error must occur.

Expressions are evaluated at request processing time. The value of an expression is converted to a String and inserted at the proper position in the .jsp file.

The <jsp:expression> element (Section JSP.6.3.7) describes the same information following the XML syntax.

Examples

This example inserts the current date.

```
<%= (new java.util.Date()).toLocaleString() %>
```

Syntax

```
<%= expression %>
```

JSP.1.13 Actions

Actions may affect the current out stream and use, modify and/or create objects. Actions may depend on the details of the specific request object received by the JSP page.

The JSP specification includes some actions that are standard and must be implemented by all conforming JSP containers; these actions are described in Chapter 5.

New actions are defined according to the mechanisms described in Chapters 7 and 13 and are introduced using the taglib directive.

The syntax for action elements is based on XML. Actions can be empty or non-empty.

JSP.1.14 Tag Attribute Interpretation Semantics

The interpretation of all actions start by evaluating the values given to its attributes left to right, and assigning the values to the attributes. In the process some conversions may be applicable; the rules for them are described in Section JSP.1.14.2.

Many values are fixed translation-time values, but JSP 2.0 also provides a mechanism for describing values that are computed at request time, the rules are described in Section JSP.1.14.1.

JSP.1.14.1 Request Time Attribute Values

An attribute value of the form "<%= scriptlet_expr %>" or '<%= scriptlet_expr %>' denotes a request-time attribute value. The value denoted is that of the scriptlet expression involved. If Expression Language evaluation is not deactivated for the translation unit (see Section JSP.3.3.2, "Deactivating EL Evaluation") then request-time attribute values can also be specified using the EL using the syntax '${el_expr}' or "${el_expr}". Containers must also recognize multiple EL expressions mixed with optional string constants. For example, "Version ${major}.${minor} Installed" is a valid request-time attribute value.

Request-time attribute values can only be used in actions. If a request-time attribute value is used in a directive, a translation error must occur. If there are more than one such attribute in a tag, the expressions are evaluated left-to-right.

Quotation is done as in any other attribute value (Section JSP.1.6).

Only attribute values can be denoted this way (the name of the attribute is always an explicit name). When using scriptlet expressions, the expression must appear by itself (multiple expressions, and mixing of expressions and string constants are not permitted). Multiple operations must be performed within the expression. Type conversions are described in Section JSP.1.14.2.

By default, except in tag files, all attributes have page translation-time semantics. Attempting to specify a scriptlet expression or EL expression as the value for an attribute that (by default or otherwise) has page translation time semantics is illegal, and will result in a fatal translation error. The type of an action element indicates whether a given attribute will accept request-time attribute values.

Most attributes in the standard actions from Chapter 5 have page translation-time semantics, but the following attributes accept request-time attribute expressions:

- The value attribute of jsp:setProperty (Section JSP.5.2).
- The beanName attribute of jsp:useBean (Section JSP.5.1).
- The page attribute of jsp:include (Section JSP.5.4).
- The page attribute of jsp:forward (Section JSP.5.5).
- The value attribute of jsp:param (Section JSP.5.6).
- The height and width attributes of jsp:plugin (Section JSP.5.7).
- The name attribute of jsp:element (Section JSP.5.14).

JSP.1.14.2 Type Conversions

We describe two cases for type conversions

JSP.1.14.2.1 Conversions from String values

A string value can be used to describe a value of a non-String type through a conversion. Whether the conversion is possible, and, if so, what is it, depends on a target type.

String values can be used to assign values to a type that has a PropertyEditor class as indicated in the JavaBeans specification. When that is the case, the setAs-

Text(String) method is used. A conversion failure arises if the method throws an IllegalArgumentException.

String values can also be used to assign to the types as listed in Table JSP.1-11. The conversion applied is that shown in the table.

A conversion failure leads to an error, whether at translation time or request-time.

Table JSP.1-11 *Conversions from string values to target type*

Target Type	Source String Value
Bean Property	As converted by the corresponding PropertyEditor, if any, using PropertyEditor.setAsText(string-literal) and PropertyEditor.getValue(). If there is no corresponding PropertyEditor or the PropertyEditor throws an exception, 'null' if the string is empty, otherwise error.
boolean or Boolean	As indicated in java.lang.Boolean.valueOf(String). This results in 'false' if the String is empty.
byte or Byte	As indicated in java.lang.Byte.valueOf(String), or '(byte) 0' if the string is empty.
char or Character	As indicated in String.charAt(0), or '(char) 0' if the string is empty.
double or Double	As indicated in java.lang.Double.valueOf(String), or 0 if the string is empty.
int or Integer	As indicated in java.lang.Integer.valueOf(String), or 0 if the string is empty.
float or Float	As indicated in java.lang.Float.valueOf(String), or 0 if the string is empty.
long or Long	As indicated in java.lang.Long.valueOf(String), or 0 if the string is empty.
short or Short	As indicated in java.lang.Short.valueOf(String), or 0 if the string is empty.
Object	As if new String(string-literal). This results in new String("") if the string is empty.

These conversions are part of the generic mechanism used to assign values to attributes of actions: when an attribute value that is not a request-time

attribute is assigned to a given attribute, the conversion described here is used, using the type of the attribute as the target type. The type of each attribute of the standard actions is described in this specification, while the types of the attributes of a custom action are described in its associated Tag Library Descriptor.

A given action may also define additional ways where type/value conversions are used. In particular, Section JSP.5.2 describes the mechanism used for the setProperty standard action.

JSP.1.14.2.2 Conversions from request-time expressions

Request-time expressions can be assigned to properties of any type. In the case of scriptlet expressions, no automatic conversions will be performed. In the case of EL expressions, the rules in Section JSP.2.8, “Type Conversion” must be followed.

CHAPTER JSP.2

Expression Language

This chapter describes the expression language used by JSP 2.0. The expressioin language is independent of JSP details except for the set of implicit objects. The language was initially defined by the JSP StandardTag Library (JSTL) 1.0 specification, but is now incorporated in the JSP specification, and extended with new features. A JSTL maintenance release (JSTL 1.1) aligns itself with the JSP 2.0 version of the language. The JavaServer Faces expert group (JSR-127) is also considering to use this expression language.

The language semantics are exposed through an API described in the javax.servlet.jsp.el package. The main use of this API is to implement the JSP 2.0 language in a JSP container, but it may be used by JSP developers, most likely tag hander authors.

Sections JSP.2.1 and JSP.2.2 describe how the expression language is used in JSP 2.0 while sections JSP.2.3 to JSP.2.9 provide the generic description of the expression language. The API to the expression language is described in full in Chapter JSP.14.

JSP.2.1 Overview

The EL is a simple language based on:

- Available namespace (the PageContext attributes)
- Nested properties and accessors to collections
- Relational, logical and arithmetic operators.
- Extensible functions mapping into static methods in Java classes.
- A set of implicit objects

The EL is inspired by both ECMAScript and the XPath expression languages. The expert groups of JSR-052 and JSR-152 were very reluctant to design yet another expression language and tried to use each of these languages but both were found to fall short in different areas. The feedback received from users of JSTL 1.0 has been very positive.

The EL is available in attribute values for standard and custom actions and within template text; in both cases the EL is invoked consistently via the construct ${expr}.

The addition of the EL to the JSP technology facilitates much the writing of script-less JSP pages. These pages can use EL expressions but can't use Java scriptlets, Java expressions, or Java declaration elements. This usage pattern can be enforced through the scripting-invalid JSP configuration element.

JSP.2.2 The Expression Language in JSP 2.0

The expression language is used in a number of places within the JSP 2.0 language.

JSP.2.2.1 Expressions and Attribute Values

EL expressions can be used in any attribute that can accept a run-time expression, be it a standard action or a custom action (see the section below on backward compatibility issues).

There are three use cases for expressions in attribute values:

- The attribute value contains a single expression construct

  ```
  <some:tag value="${expr}"/>
  ```

  In this case, the expression is evaluated and the result is coerced to the attribute's expected type according to the type conversion rules described later.

- The attribute value contains one or more expressions separated or surrounded by text:

  ```
  <some:tag value="some${expr}${expr}text${expr}"/>
  ```

  In this case, the expressions are evaluated from left to right, coerced to Strings (according to the type conversion rules described later), and concatenated

with any intervening text. The resulting String is then coerced to the attribute's expected type according to the type conversion rules described later.

- The attribute value contains only text:

```
<some:tag value="sometext"/>
```

In this case, the attribute's String value is coerced to the attribute's expected type according to the type conversion rules described in Section JSP.2.8. These rules are equivalent to the JSP 1.2 conversions, except that empty strings are treated differently.

JSP.2.2.1.1 Examples

The following shows a conditional action that uses the EL to test whether a property of a bean is less than 3.

```
<c:if test="${bean1.a < 3}">
...
</c:if>
```

Note that the normal JSP coercion mechanism already allows for:

```
<mytags:if test="true" />
```

There may be literal values that include the character sequence ${. If that is the case, a literal with that value can be used as shown here:

```
<mytags:example code="an expression is ${'${'}expr}" />
```

The resulting attribute value would then be the string an expression is ${expr}.

JSP.2.2.2 Expressions and Template Text

The EL can be used directly in template text, be it inside the body of a custom or standard actions or in template text outside of any action. Exceptions are if the body of the tag is tagdependent, or if EL is turned off (usually for compatibility issues) explicitly through a directive or implicitly; see below.

The semantics of an EL expression are the same as with Java expressions: the value is computed and inserted into the current output. In cases where escaping is desired (for example, to help prevent cross-site scripting attacks), the JSTL core tag <c:out> can be used. For example:

```
<c:out value="${anELexpression}" />
```

JSP.2.2.2.1 Examples

The following shows a custom action where two EL expressions are used to access bean properties:

```
<c:wombat>
One value is ${bean1.a} and another is ${bean2.a.c}
</c:wombat>
```

JSP.2.2.3 Implicit Objects

There are several implicit objects that are available to EL expressions used in JSP pages. These objects are always available under these names:

- pageContext - the PageContext object
- pageScope - a Map that maps page-scoped attribute names to their values
- requestScope - a Map that maps request-scoped attribute names to their values
- sessionScope - a Map that maps session-scoped attribute names to their values
- applicationScope - a Map that maps application-scoped attribute names to their values
- param - a Map that maps parameter names to a single String parameter value (obtained by calling ServletRequest.getParameter(String name))
- paramValues - a Map that maps parameter names to a String[] of all values for that parameter (obtained by calling ServletRequest.getParameterValues(String name))
- header - a Map that maps header names to a single String header value (obtained by calling ServletRequest.getHeader(String name))
- headerValues - a Map that maps header names to a String[] of all values for that header (obtained by calling ServletRequest.getHeaders(String))
- cookie - a Map that maps cookie names to a single Cookie object. Cookies are retrieved according to the semantics of HttpServletRequest.getCookies(). If the same name is shared by multiple cookies, an implementation must use the first one encountered in the array of `Cookie` objects returned by the getCookies() method. However, users of the cookie implicit object must be aware that the ordering of cookies is currently unspecified in the servlet specification.

- initParam - a Map that maps context initialization parameter names to their String parameter value (obtained by calling ServletContext.getInitParameter(String name))

The following table shows some examples of using these implicit objects:

Table JSP.2-1 *Examples of Using Implicit Objects*

Expression	Result
${pageContext.request.requestURI}	The request's URI (obtained from HttpServletRequest)
${sessionScope.profile}	The session-scoped attribute named profile (null if not found)
${param.productId}	The String value of the productId parameter, or null if not found
${paramValues.productId}	The String[] containing all values of the productId parameter, or null if not found

JSP.2.2.4 Deactivating EL Evaluation

Since the syntactic pattern ${*expr*} was not reserved in the JSP specifications before JSP 2.0, there may be situations where such a pattern appears but the intention is not to activate EL expression evaluation but rather to pass through the pattern verbatim. To address this, the EL evaluation machinery can be deactivated as indicated in Section JSP.3.3.2, “Deactivating EL Evaluation”.

JSP.2.2.5 Disabling Scripting Elements

With the addition of the EL, some JSP page authors, or page authoring groups, may want to follow a methodology where scripting elements are not allowed. See Section JSP.3.3.3, “Disabling Scripting Elements” for more details.

JSP.2.3 General Syntax of the Expression Language

JSP containers are required to produce a translation error when a syntactically invalid EL expression is encountered in an attribute value or within template text. The syntax of an EL expression is described in detail in this section.

JSP.2.3.1 Overview

The syntax is quite simple. Variables are accessed by name. A generalized [] operator can be used to access maps, lists, arrays of objects and properties of a JavaBeans object; the operator can be nested arbitrarily. The . operator can be used as a convenient shorthand for property access when the property name follows the conventions of Java identifiers, but the [] operator allows for more generalized access.

Relational comparisons are allowed using the standard Java relational operators. Comparisons may be made against other values, or against boolean (for equality comparisons only), String, integer, or floating point literals. Arithmetic operators can be used to compute integer and floating point vlaues. Logical operators are available.

JSP.2.3.2 Literals

There are literals for boolean, integer, floating point, string, null.

- Boolean - true and false
- Integer - As defined by the IntegerLiteral construct in Section JSP.2.9
- Floating point - As defined by the FloatingPointLiteral construct in Section JSP.2.9
- String - With single and double quotes - " is escaped as \", ' is escaped as \', and \ is escaped as \\. Quotes only need to be escaped in a string value enclosed in the same type of quote
- Null - null

JSP.2.3.3 Errors, Warnings, Default Values

JSP pages are mostly used in presentation, and in that usage, experience suggests that it is most important to be able to provide as good a presentation as possible, even when there are simple errors in the page. To meet this requirement, the EL does not provide warnings, just default values and errors. Default values are type-correct values that are assigned to a subexpression when there is some problem. An error is an exception thrown (to be handled by the standard JSP machinery).

JSP.2.3.4 Operators "[]" and "."

The EL follows ECMAScript in unifying the treatment of the . and [] operators.

expr-a.identifier-b is equivalent to expr-a["identifier-b"]; that is, the identifier identifier-b is used to construct a literal whose value is the identifier, and then the [] operator is used with that value.

To evaluate expr-a[expr-b]:

- Evaluate expr-a into value-a
- If value-a is null, return null.
- Evaluate expr-b into value-b
- If value-b is null, return null.
- If value-a is a Map, List, or array:
  - If value-a is a Map:
    - If !value-a.containsKey(value-b) then return null.
    - Otherwise, return value-a.get(value-b)
  - If value-a is a List or array:
    - Coerce value-b to int (using coercion rules)
      - If coercion couldn't be performed: error
    - Then, if value-a.get(value-b) or Array.get(value-a, value-b) throws ArrayIndexOutOfBoundsException or IndexOutOfBoundsException: return null
    - Otherwise, if value-a.get(value-b) or Array.get(value-a, value-b) throws other exception, error
    - Otherwise, return value-a.get(value-b) or Array.get(value-a, value-b), as appropriate.
- Otherwise (a JavaBeans object), coerce value-b to String
  - If value-b is a readable property of value-a, as per the JavaBeans specification:
    - If getter throws an exception: error
    - Otherwise: return result of getter call
  - Otherwise: error.

JSP.2.3.5 Arithmetic Operators

Arithmetic is provided to act on integer (BigInteger and Long) and floating point (BigDecimal and Double) values. There are 5 operators:

- Addition: +
- Substraction: -

- Multiplication: *
- Division: / and div
- Remainder (modulo): % and mod

The last two operators are available in both syntaxes to be consistent with XPath and ECMAScript.

The evaluation of arithmetic operators is described in the following sections. A and B are the evaluation of subexpressions

JSP.2.3.5.1 Binary operators - A {+,-,*} B

- If A and B are null, return (Long) 0
- If A or B is a BigDecimal, coerce both to BigDecimal and then:
  - If operator is +, return A.add(B)
  - If operator is -, return A.subtract(B)
  - If operator is *, return A.multiply(B)
- If A or B is a Float, Double, or String containing ., e, or E:
  - If A or B is BigInteger, coerce both A and B to BigDecimal and apply operator.
  - Otherwise, coerce both A and B to Double and apply operator
- If A or B is BigInteger, coerce both to BigInteger and then:
  - If operator is +, return A.add(B)
  - If operator is -, return A.subtract(B)
  - If operator is *, return A.multiply(B)
- Otherwise coerce both A and B to Long and apply operator
- If operator results in exception, error

JSP.2.3.5.2 Binary operator - A {/,div} B

- If A and B are null, return (Long) 0
- If A or B is a BigDecimal or a BigInteger, coerce both to BigDecimal and return A.divide(B, BigDecimal.ROUND_HALF_UP).
- Otherwise, coerce both A and B to Double and apply operator
- If operator results in exception, error

JSP.2.3.5.3 Binary operator - A {%,mod} B

- If A and B are null, return (Long) 0
- If A or B is a BigDecimal, Float, Double, or String containing ., e, or E, coerce both A and B to Double and apply operator
- If A or B is a BigInteger, coerce both to BigInteger and return A.remainder(B).
- Otherwise coerce both A and B to Long and apply operator
- If operator results in exception, error

JSP.2.3.5.4 Unary minus operator - -A

- If A is null, return (Long) 0
- If A is a BigDecimal or BigInteger, return A.negate().
- If A is a String:
  - If A contains ., e, or E, coerce to a Double and apply operator
  - Otherwise, coerce to a Long and apply operator
  - If operator results in exception, error
- If A is Byte, Short, Integer, Long, Float, Double
  - Retain type, apply operator
  - If operator results in exception, error
- Otherwise, error

JSP.2.3.5.5 Relational Operators

The relational operators are:

- == and eq
- != and ne
- < and lt
- > and gt
- <= and le
- >= and ge

The second versions of the last 4 operators are made available to avoid having to use entity references in XML syntax and have the exact same behavior, i.e. < behaves the same as lt and so on.

The evaluation of relational operators is described in the following sections.

JSP.2.3.5.6 A {<,>,<=,>=,lt,gt,le,ge} B

- If A==B, if operator is <=, le, >=, or ge return true. Otherwise return false
- If A is null or B is null, return false
- If A or B is BigDecimal, coerce both A and B to BigDecimal and use the return value of A.compareTo(B).
- If A or B is Float or Double coerce both A and B to Double apply operator
- If A or B is BigInteger, coerce both A and B to BigInteger and use the return value of A.compareTo(B).
- If A or B is Byte, Short, Character, Integer, or Long coerce both A and B to Long and apply operator
- If A or B is String coerce both A and B to String, compare lexically
- If A is Comparable, then:
  - If A.compareTo (B) throws exception, error.
  - Otherwise use result of A.compareTo(B)
- If B is Comparable, then:
  - If B.compareTo (A) throws exception, error.
  - Otherwise use result of B.compareTo(A)
- Otherwise, error

JSP.2.3.5.7 A {==,!=,eq,ne} B

- If A==B, apply operator
- If A is null or B is null return false for == or eq, true for != or ne.
- If A or B is BigDecimal, coerce both A and B to BigDecimal and then:
  - If operator is == or eq, return A.equals(B)
  - If operator is != or ne, return !A.equals(B)
- If A or B is Float or Double coerce both A and B to Double, apply operator

- If A or B is BigInteger, coerce both A and B to BigInteger and then:
  - If operator is == or eq, return A.equals(B)
  - If operator is != or ne, return !A.equals(B)
- If A or B is Byte, Short, Character, Integer, or Long coerce both A and B to Long, apply operator
- If A or B is Boolean coerce both A and B to Boolean, apply operator
- If A or B is String coerce both A and B to String, compare lexically
- Otherwise if an error occurs while calling A.equals(B), error
- Otherwise, apply operator to result of A.equals(B)

JSP.2.3.6 Logical Operators

The logical operators are:

- && and and
- || and or
- ! and not

The evaluation of logical operators is described in the following sections.

JSP.2.3.6.1 Binary operator - A {&&,||,and,or} B

- Coerce both A and B to Boolean, apply operator

The operator stops as soon as the expression can be determined, i.e., A and B and C and D – if B is false, then only A and B is evaluated.

JSP.2.3.6.2 Unary not operator - {!,not} A

- Coerce A to Boolean, apply operator

JSP.2.3.7 Empty Operator - empty A

The empty operator is a prefix operator that can be used to determine if a value is null or empty.

To evaluate empty A

- If A is null, return true,
- Otherwise, if A is the empty string, then return true.
- Otherwise, if A is an empty array, then return true.
- Otherwise, if A is an empty Map, return true,
- Otherwise, if A is an empty Collection, return true,
- Otherwise return false.

JSP.2.3.8 Conditional Operator - A ? B : C

Evaluate B or C, depending on the result of the evaluation of A.

- Coerce A to Boolean:
  - If A is true, evaluate and return B
  - If A is false, evaluate and return C

JSP.2.3.9 Parentheses

Parentheses can be used to change precedence, as in: ${ (a * (b + c)) }

JSP.2.3.10 Operator Precedence

Highest to lowest, left-to-right.

- [] .
- ()
- - (unary) not ! empty
- * / div % mod
- + - (binary)
- < > <= >= lt gt le ge
- == != eq ne
- && and
- || or
- ? :

JSP.2.4 Reserved Words

The following words are reserved for the language and should not be used as identifiers.

and	eq	gt	true	instanceof	
or	ne	le	false	empty	
not	lt	ge	null	div	mod

Note that many of these words are not in the language now, but they may be in the future, so developers should avoid using these words now.

JSP.2.5 Named Variables

A core concept in the EL is the evaluation of a variable name into an object. The EL API provides a generalized mechanism, a VariableResolver, that will resolve names into objects. The default resolver is what is used in the evaluation of EL expressions in template and attributes. This default resolver provides the implicit objects mentioned in Section JSP.2.2.3. The default resolver also provides a map for other identifiers by looking up its value as an attribute, according to the behavior of PageContext.findAttribute(String) on the pageContext object. For example:

```
${product}
```

This expression will look for the attribute named product, searching the page, request, session, and application scopes, and will return its value. If the attribute is not found, null is returned.

Note that an identifier that matches one of the implicit objects described in the next section will return that implicit object instead of an attribute value.

See Chapter JSP.14 for further details on the VariableResolver and how it fits with the evaluation API.

JSP.2.6 Functions

The EL has qualified functions, reusing the notion of qualification from XML namespaces (and attributes), XSL functions, and JSP custom actions. Functions are mapped to public static methods in Java classes. In JSP 2.0 the map is specified in the TLD.

JSP.2.6.1 Invocation Syntax

The full syntax is that of qualified n-ary functions:

```
ns:f(a1,a2, ..., an)
```

As with the rest of EL, this element can appear in attributes and directly in template text.

The prefix ns must match the prefix of a tag library that contains a function whose name and signature matches the function being invoked (f), or a translation error must occur. If the prefix is omitted, the tag library associated with the default namespace is used (this is only possible in JSP documents).

In the following standard syntax example, func1 is associated with some-taglib:

```
<%@ taglib prefix="some" uri="http://acme.com/some-taglib" %>
${some:func1(true)}
```

In the following JSP document example, both func2 and func3 are associated with default-taglib:

```
<some:tag   xmlns="http://acme.com/default-taglib"
            xmlns:some="http://acme.com/some-taglib"
            xmlns:jsp="http://java.sun.com/JSP/Page">
    <some:other value="${func2(true)}">
        ${func3(true)}
    </some:other>
</some:tag>
```

JSP.2.6.2 Tag Library Descriptor Information

Each tag library may include zero or more n-ary (static) functions. The Tag Library Descriptor (TLD) associated with a tag library lists the functions.

Each such function is given a name (as seen in the EL), and a static method in a specific class that will implement the function. The class specified in the TLD must be a public class, and must be specified using a fully-qualified class name (including packages). The specified method must be a public static method in the specified class, and must be specified using a fully-qualified return type followed by the method name, followed by the fully-qualified argument types in parenthesis, separated by commas (see the XML Schema in Appendix JSP.C for a full description of this syntax). Failure to satisfy these requirements shall result in a translation-time error.

A tag library can have only one function element in the same tag library with the same value for their name element. If two functions have the same name, a translation-time error shall be generated.

Reference the function element in Section JSP.C.1, “XML Schema for TLD, JSP 2.0” for how to specify a function in the TLD.

JSP.2.6.3 Example

The following TLD fragment describes a function with name nickname that is intended to fetch the nickname of the user:

```
<taglib>
    ...
    <function>
        <name>nickname</name>
        <function-class>mypkg.MyFunctions</function-class>
        <function-signature>
            java.lang.String nickName(java.lang.String)
        </function-signature>
    </function>
</taglib>
```

The following EL fragment shows the invocation of the function:

```
<h2>Dear ${my:nickname(user)}</h2>
```

JSP.2.6.4 Semantics

- If the function has no prefix, the default namespace is used. If the function has a prefix, assume the namespace as that associated with the prefix.

  Let ns be the namespace associated with the function, and f be the name of the function.

- Locate the TLD associated with ns. If none can be found, this shall be a translation-time error.
- Locate the function element with a name subelement with value f in that TLD. If none can be found, this shall be a translation-time error.
- Locate the public class with name equal to the value of the function-class element. Locate the public static method with name and signature equal to the

value of the function-signature element. If any of these don't exist, a translation-time error shall occur..

- Evaluate each argument to the corresponding type indicated in the signature
- Evaluate the public static Java method. The resulting value is the value returned by the method evaluation, or null if the Java method is declared to return void. If an exception is thrown during the method evaluation, the exception must be wrapped in an ELException and the ELException must be thrown.

JSP.2.7 Implicit Objects

The EL defines a set of implicit objects which depends on the context in which the EL is being used. When an expression references one of these objects by name, the appropriate object is returned instead of the corresponding attribute. For example in the context of JSP pages, ${pageContext} returns the PageContext object, even if there is an existing pageContext attribute containing some other value. See Section JSP.2.2.3 for details.

JSP.2.8 Type Conversion

Every expression is evaluated in the context of an expected type. The result of the expression evaluation may not match the expected type exactly, so the rules described in the following sections are applied:

JSP.2.8.1 To Coerce a Value X to Type Y

- If X is of a primitive type, Let X' be the equivalent "boxed form" of X. Otherwise, Let X' be the same as X.
- If Y is of a primitive type, Let Y' be the equivalent "boxed form" of Y. Otherwise, Let Y' be the same as Y.
- Apply the rules in Sections JSP.2.8.2-JSP.2.8.6 for coercing X' to Y'.
- If Y is a primitive type, then the result is found by "unboxing" the result of the coercion. If the result of the coercion is null, then error.
- If Y is not a primitive type, then the result is the result of the coercion.

For example, if coercing an int to a String, "box" the int into an Integer and apply the rule for coercing an Integer to a String. Or if coercing a String to a double, apply the rule for coercing a String to a Double, then "unbox" the resulting Double, making sure the resulting Double isn't actually `null`.

JSP.2.8.2 Coerce A to String

- If A is String: return A
- Otherwise, if A is null: return ""
- Otherwise, if A.toString() throws an exception, error
- Otherwise, return A.toString()

JSP.2.8.3 Coerce A to Number type N

- If A is null or "", return 0.
- If A is Character, convert A to new Short((short)a.charValue()), and apply the following rules.
- If A is Boolean, then error.
- If A is Number type N, return A
- If A is Number, coerce quietly to type N using the following algorithm:
  - If N is BigInteger
    - If A is a BigDecimal, return A.toBigInteger()
    - Otherwise, return BigInteger.valueOf(A.longValue())
  - If N is BigDecimal,
    - If A is a BigInteger, return new BigDecimal(A)
    - Otherwise, return new BigDecimal(A.doubleValue())
  - If N is Byte, return new Byte(A.byteValue())
  - If N is Short, return new Short(A.shortValue())
  - If N is Integer, return new Integer(A.intValue())
  - If N is Long, return new Long(A.longValue())
  - If N is Float, return new Float(A.floatValue())
  - If N is Double, return new Double(A.doubleValue())
  - Otherwise, error.

- If A is String, then:
  - If N is BigDecimal then:
    - If new BigDecimal(A) throws an exception then error.
    - Otherwise, return new BigDecimal(A).
  - If N is BigInteger then:
    - If new BigInteger(A) throws an exception then error.
    - Otherwise, return new BigInteger(A).
  - If N.valueOf(A) throws an exception, then error.
  - Otherwise, return N.valueOf(A).
- Otherwise, error.

JSP.2.8.4 Coerce A to Character

- If A is null or "", return (char) 0
- If A is Character, return A
- If A is Boolean, error
- If A is Number, coerce quietly to type Short, then return a Character whose numeric value is equivalent to that of a Short.
- If A is String, return A.charAt (0)
- Otherwise, error

JSP.2.8.5 Coerce A to Boolean

- If A is null or "", return false
- Otherwise, if A is a Boolean, return A
- Otherwise. if A is a String, and Boolean.valueOf(A) does not throw an exception, return it
- Otherwise, error

JSP.2.8.6 Coerce A to Any Other Type T

- If A is null, return null
- If A is assignable to T, coerce quietly

- If A is a String, and T has no PropertyEditor:
  - If A is "", return null
  - Otherwise error
- If A is a String and T's PropertyEditor throws an exception:
  - If A is "", return null
  - Otherwise, error
- Otherwise, apply T's PropertyEditor
- Otherwise, error

JSP.2.9 Collected Syntax

The following are the constructs supported by the EL:

```
Expression          ::=  Expression1 ExpressionRest?

ExpressionRest      ::=  ‘?’ Expression ‘:’ Expression

Expression1         ::=      Expression BinaryOp Expression
                        |    UnaryExpression
```

```
BinaryOp            ::=   'and'
                        |  ‘&&’
                        |  ‘or’
                        |  ‘||’
                        |  '+'
                        |  '-'
                        |  '*'
                        |  '/'
                        |  'div'
                        |  '%'
                        |  'mod'
                        |  '>'
                        |  'gt'
                        |  '<'
                        |  'lt'
                        |  '>='
                        |  'ge'
                        |  '<='
                        |  'le'
                        |  '=='
                        |  ‘eq’
                        |  ‘!=’
                        |  ‘ne’

UnaryExpression     ::=   UnaryOp UnaryExpression
                        |  Value

UnaryOp             ::=   '-'
                        |  ‘!’
                        |  ‘not’
                        |  ‘empty’

Value               ::=   ValuePrefix
                        |  Value ValueSuffix

ValuePrefix         ::=   Literal
                        |  '(' Expression ')'
                        |  Identifier except for ImplicitObject
                        |  ImplicitObject
                        |  FunctionInvocation

ValueSuffix         ::=   ‘.’ Identifier
                        |  ‘[‘ Expression ‘]’

Identifier          ::=   Java language identifier
```

```
ImplicitObject      ::=     'pageContext'
                        |   ‘pageScope’
                        |   'requestScope'
                        |   'sessionScope'
                        |   'applicationScope'
                        |   'param'
                        |   'paramValues'
                        |   'header'
                        |   'headerValues'
                        |   'initParam'
                        |   'cookie'

FunctionInvocation  ::= (Identifier ‘:’)? Identifier ‘(‘
                        ( Expression ( ‘,’ Expression )* )? ‘)’

Literal             ::=     BooleanLiteral
                        |   IntegerLiteral
                        |   FloatingPointLiteral
                        |   StringLiteral
                        |   NullLiteral

BooleanLiteral      ::=     'true'
                        |   ‘false’

StringLiteral       ::=     '([^'\]|\'|\\)*'
                        |   "([^"\]|\"|\\)*"
                        i.e., a string of any characters enclosed by single
                        or double quotes, where \ is used to escape ', ",
                        and \. It is possible to use single quotes within
                        double quotes, and vice versa, without escaping.

IntegerLiteral      ::= [‘0’-’9’]+

FloatingPointLiteral::=     ([‘0’-’9’])+ ‘.’ ([‘0’-’9’])* Exponent?
                        |   ‘.’ ([‘0’-’9’])+ Exponent?
                        |   ([‘0’-’9’])+ Exponent?

Exponent            ::= [‘e’,’E’] ([‘+’,’-’])? ([‘0’-’9’])+

NullLiteral         ::= 'null'
```

Notes

- An identifier is constrained to be a Java identifier - e.g., no -, no /, etc.
- A String only recognizes a limited set of escape sequences, and \ may not appear unescaped.
- The relational operator for equality is == (double equals).
- The value of an IntegerLiteral ranges from Long.MIN_VALUE to Long.MAX_VALUE
- The value of a FloatingPointLiteral ranges from Double.MIN_VALUE to Double.MAX_VALUE

CHAPTER JSP.3

JSP Configuration

This chapter describes the JSP configuration information, which is specified in the Web Application Deployment Descriptor in WEB-INF/web.xml. As of Servlet 2.4, the Web Application Deployment Descriptor is defined using XML Schema, and imports the elements described in Section JSP.B.1, "XML Schema for JSP 2.0 Deployment Descriptor". See that section for the details on how to specify JSP configuration information in a Web Application.

JSP.3.1 JSP Configuration Information in web.xml

A Web Application can include general JSP configuration information in its web.xml file that is to be used by the JSP container. The information is described through the jsp-config element and its subelements.

The jsp-config element is a subelement of web-app that is used to provide global configuration information for the JSP files in a Web Application. A jsp-config has two subelements: taglib and jsp-property-group, defining the taglib mapping and groups of JSP files respectively.

JSP.3.2 Taglib Map

The web.xml file can include an explicit taglib map between URIs and TLD resource paths described using taglib elements in the Web Application Deployment descriptor.

The taglib element is a subelement of jsp-config that can be used to provide information on a tag library that is used by a JSP page within the Web Application. The taglib element has two subelements: taglib-uri and taglib-location.

A taglib-uri element describes a URI identifying a tag library used in the web application. The body of the taglib-uri element may be either an absolute URI specification, or a relative URI as in Section JSP.1.2.1. There should be no entries in web.xml with the same taglib-uri value.

A taglib-location element contains a resource location (as indicated in Section JSP.1.2.1) of the Tag Library Description File for the tag library.

JSP.3.3 JSP Property Groups

A JSP property group is a collection of properties that apply to a set of files that represent JSP pages. These properties are defined in one or more jsp-property-group elements in the Web Application deployment descriptor.

Most properties defined in a JSP property group apply to an entire translation unit, that is, the requested JSP file that is matched by its URL pattern and all the files it includes via the include directive. The exception is the page-encoding property, which applies separately to each JSP file matched by its URL pattern.

The applicability of a JSP property group is defined through one or more URL patterns. URL patterns use the same syntax as defined in Chapter SRV.11 of the Servlet 2.4 specification, but are bound at translation time. All the properties in the group apply to the resources in the Web Application that match any of the URL patterns. There is an implicit property: that of being a JSP file. JSP Property Groups do not affect tag files.

If a resource matches a URL pattern in both a <servlet-mapping> and a <jsp-property-group>, the pattern that is most specific applies (following the same rules as in the Servlet specification). If the URL patterns are identical, the <jsp-property-group> takes precedence over the <servlet-mapping>. If at least one <jsp-property-group> contains the most specific matching URL pattern, the resource is considered to be a JSP file, and the properties in that <jsp-property-group> apply. In addition, if a resource is considered to be a JSP file, all include-prelude and include-coda properties apply from all the <jsp-property-group> elements with matching URL patterns (see Section JSP.3.3.5).

JSP.3.3.1 JSP Property Groups

A jsp-property-group is a subelement of jsp-config. The properties that can currently be described in a jsp-property-group include:

- Indicate that a resource is a JSP file (implicit).
- Control disabling of EL evaluation.
- Control disabling of Scripting elements.
- Indicate page Encoding information.
- Prelude and Coda automatic includes.
- Indicate that a resource is a JSP document.

JSP.3.3.2 Deactivating EL Evaluation

Since the syntactic pattern ${*expr*} was not reserved in the JSP specifications before JSP 2.0, there may be situations where such a pattern appears but the intention is not to activate EL expression evaluation but rather to pass through the pattern verbatim. To address this, the EL evaluation machinery can be deactivated as indicated in this section.

Each JSP page has a default setting as to whether to ignore EL expressions. When ignored, the expression is passed through verbatim. The default setting does not apply to tag files, which always default to evaluating expressions.

The default mode for JSP pages in a Web Application delivered using a web.xml using the Servlet 2.3 or earlier format is to ignore EL expressions; this provides for backward compatibility.

The default mode for JSP pages in a Web Application delivered using a web.xml using the Servlet 2.4 format is to evaluate EL expressions; this automatically provides the default that most applications want.

The default mode can be explicitly changed by setting the value of the el-ignored element. The el-ignored element is a subelement of jsp-property-group (see Section JSP.3.3.1, “JSP Property Groups”). It has no subelements. Its valid values are true and false.

For example, the following web.xml fragment defines a group that deactivates EL evaluation for all JSP pages delivered using the .jsp extension:

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

Page authors can override the default mode through the isELIgnored attribute of the page directive. For tag files, there is no default, but the isELIgnored attribute of the tag directive can be used to control the EL evaluation settings.

Table JSP.3-1 summarizes the EL evaluation settings for JSP pages, and their meanings:

Table JSP.3-1 *EL Evaluation Settings for JSP Pages*

JSP Configuration <el-ignored>	Page Directive isELIgnored	EL Encountered
unspecified	unspecified	Ignored if <= 2.3 web.xml Evaluated otherwise.
false	unspecified	Evaluated
true	unspecified	Ignored
don't care	false	Evaluated
don't care	true	Ignored

Table JSP.3-2 summarizes the EL evaluation settings for tag files, and their meanings:

Table JSP.3-2 *EL Evaluation Settings for Tag Files*

Tag Directive isELIgnored	EL Encountered
unspecified	Evaluated
false	Evaluated
true	Ignored

The EL evaluation setting for a translation unit also affects whether the \$ quote sequence is enabled for template text and attribute values in a JSP page, document, or tag file. When EL evaluation is disabled, \$ will not be recognized as a quote, whereas when EL evaluation is enabled, \$ will be recognized as a quote for $. See Section JSP.1.6, "Quoting and Escape Conventions" and Section JSP.6.2.2, "Overview of Syntax of JSP Documents" for details.

JSP.3.3.3 Disabling Scripting Elements

With the addition of the EL, some JSP page authors, or page authoring groups, may want to follow a methodology where scripting elements are not allowed. Previous versions of JSP enabled this through the notion of a TagLibraryValidator that would verify that the elements are not present. JSP 2.0 makes this slightly easier through a JSP configuration element.

The scripting-invalid element is a subelement of jsp-property-group (see 3.3.1). It has no subelements. Its valid values are true and false. Scripting is enabled by default. Disabling scripting elements can be done by setting the scripting-invalid element to true in the JSP configuration.

For example, the following web.xml fragment defines a group that disables scripting elements for all JSP pages delivered using the .jsp extension:

```
<jsp-property-group>
    <url-pattern>*.jsp</url-pattern>
    <scripting-invalid>true</scripting-invalid>
</jsp-property-group>
```

Table JSP.3-3 summarizes the scripting settings and their meanings:

Table JSP.3-3 *Scripting Settings*

JSP Configuration <scripting-invalid>	Scripting Encountered
unspecified	Valid
false	Valid
true	Translation Error

JSP.3.3.4 Declaring Page Encodings

The JSP configuration element page-encoding can be used to easily set the pageEncoding property of a group of JSP pages defined using the jsp-property-group element. This is only needed for pages in standard syntax, since for documents in XML syntax the page encoding is determined as described in section 4.3.3 and appendix F.1 of the XML specification.

The page-encoding element is a subelement of jsp-property-group (see 3.3.1). It has no subelements. Its valid values are those of the pageEncoding page

directive. It is a translation-time error to name different encodings in the pageEncoding attribute of the page directive of a JSP page and in a JSP configuration element matching the page. It is also a translation-time error to name different encodings in the prolog / text declaration of the document in XML syntax and in a JSP configuration element matching the document. It is legal to name the same encoding through multiple mechanisms.

For example, the following web.xml fragment defines a group that explicitly assigns Shift_JIS to all JSP pages and included JSP segments in the /ja subdirectory of the web application:

```
<jsp-property-group>
    <url-pattern>/ja/*</url-pattern>
    <page-encoding>Shift_JIS</page-encoding>
</jsp-property-group>
```

JSP.3.3.5 Defining Implicit Includes

The include-prelude element is an optional subelement of jsp-property-group. It has no subelements. Its value is a context-relative path that must correspond to an element in the Web Application. When the element is present, the given path will be automatically included (as in an include directive) at the beginning of the JSP page in the jsp-property-group. When there are more than one include-prelude element in a group, they are to be included in the order they appear. When more than one jsp-property-group applies to a JSP page, the corresponding include-prelude elements will be processed in the same order as they appear in the JSP configuration section of web.xml.

The include-coda element is an optional subelement of jsp-property-group. It has no subelements. Its value is a context-relative path that must correspond to an element in the Web Application. When the element is present, the given path will be automatically included (as in an include directive) at the end of the JSP page in the jsp-property-group. When there are more than one include-coda element in a group, they are to be included in the order they appear. When more than one jsp-property-group applies to a JSP page, the corresponding include-coda elements will be processed in the same order as they appear in the JSP configuration section of web.xml. Note that these semantics are in contrast to the way url-patterns are matched for other configuration elements.

Preludes and codas follow the same rules as statically included JSP segments. In particular, start tags and end tags must appear in the same file (see Section JSP.1.3.3).

For example, the following web.xml fragment defines two groups. Together they indicate that everything in directory /two/ have /WEB-INF/jspf/prelude1.jspf and /WEB-INF/jspf/prelude2.jspf at the beginning and /WEB-INF/jspf/coda1.jspf and /WEB-INF/jspf/coda2.jspf at the end, in that order, while other .jsp files only have /WEB-INF/jspf/prelude1.jspf at the beginning and /WEB-INF/jspf/coda1.jspf at the end.

```
<jsp-property-group>
    <url-pattern>*.jsp</url-pattern>
    <include-prelude>/WEB-INF/jspf/prelude1.jspf</include-prelude>
    <include-coda>/WEB-INF/jspf/coda1.jspf</include-coda>
</jsp-property-group>

<jsp-property-group>
    <url-pattern>/two/*</url-pattern>
    <include-prelude>/WEB-INF/jspf/prelude2.jspf</include-prelude>
    <include-coda>/WEB-INF/jspf/coda2.jspf</include-coda>
</jsp-property-group>
```

JSP.3.3.6 Denoting XML Documents

The JSP configuration element is-xml can be used to denote that a group of files are JSP documents, and thus must be interpreted as XML documents.

The is-xml element is a subelement of jsp-property-group (see 3.3.1). It has no subelements. Its valid values are true and false. When false, the files in the associated property group are assumed to not be JSP documents, unless there is another property group that indicates otherwise. The files are still considered to be JSP pages due to the implicit property given by the <jsp-property-group> element.

For example, the following web.xml fragment defines two groups. The first one indicates that those files with extension .jspx, which is the default extension for JSP documents, are instead just plain JSP pages. The last group indicates that files with extension .svg are actually JSP documents (which most likely are generating SVG files).

```
<jsp-property-group>
    <url-pattern>*.jspx</url-pattern>
    <is-xml>false</is-xml>
</jsp-property-group>
```

```
<jsp-property-group>
    <url-pattern>*.svg</url-pattern>
    <is-xml>true</is-xml>
</jsp-property-group>
```

CHAPTER JSP.4

Internationalization Issues

This chapter describes requirements for internationalization with JavaServer Pages 2.0 (JSP 2.0).

The JSP specification by itself does not provide a complete platform for internationalization. It is complemented by functionality provided by the underlying Java 2 Standard Edition platform, the Servlet APIs, and by tag libraries such as the JSP Standard Tag Library (JSTL) with its collection of internationalization and formatting actions. For complete information, see the respective specifications. References to JSTL are informational - this library is not required by the JSP 2.0 specification.

Primarily, this specification addresses the issues of character encodings.

The Java programming language represents characters internally using the Unicode character encoding, which provides support for most languages. As of J2SE 1.4, the Unicode 3.0 character set is supported. For storage and transmission over networks, however, many other character encodings are used. The J2SE platform therefore also supports character conversion to and from other character encodings. Any Java runtime must support the Unicode transformations UTF-8, UTF-16BE, and UTF-16LE as well as the ISO-8859-1 (Latin-1) character encoding, but most implementations support many more. The character encodings supported by Sun's Java 2 Runtime Environment version 1.3 and version 1.4 respectively are described at:

http://java.sun.com/j2se/1.3/docs/guide/intl/encoding.doc.html
http://java.sun.com/j2se/1.4/docs/guide/intl/encoding.doc.html

In JSP pages and in JSP configuration elements, character encodings are named using the names defined in the IANA charset registry:

http://www.iana.org/assignments/character-sets

JSP.4.1 Page Character Encoding

The page character encoding is the character encoding in which the JSP page or tag file itself is encoded. The character encoding is determined for each file separately, even if one file includes another using the include directive (Section JSP.1.10.3). A detailed algorithm for determining the page character encoding of a JSP page or tag file can be found in Appendix JSP.D.

For JSP pages in standard syntax, the page character encoding is determined from the following sources:

- A JSP configuration element page-encoding value whose URL pattern matches the page.
- The pageEncoding attribute of the page directive of the page. It is a translation-time error to name different encodings in the pageEncoding attribute of the page directive of a JSP page and in a JSP configuration element whose URL pattern matches the page.
- The charset value of the contentType attribute of the page directive. This is used to determine the page character encoding if neither a JSP configuration element page-encoding nor the pageEncoding attribute are provided.
- If none of the above is provided, ISO-8859-1 is used as the default character encoding.

For tag files in standard syntax, the page character encoding is determined from the pageEncoding attribute of the tag directive of the tag file, or is ISO-8859-1 if the pageEncoding attribute is not specified.

The pageEncoding and contentType attributes determine the page character encoding of only the file that physically contains them. Parsers are only required to take these attributes into consideration for character encoding detection if the directive appears at the beginning of the page or tag file and if the character encoding is an extension of ASCII, that is, if byte values 0 to 127 have the same meaning as in ASCII, at least until the attributes are found. For character encodings where this is not the case (including UTF-16 and EBCDIC-based encodings), the JSP configuration element page-encoding should be used.

For JSP documents in XML syntax, the page character encoding is determined as described in section 4.3.3 and appendix F.1 of the XML specification. It is legal to also describe the character encoding in a JSP

configuration element page-encoding or a pageEncoding attribute of the page directive of the document, as long as they are consistent. It is a translation-time error to name different encodings in two or more of the following: the XML prolog / text declaration of a JSP document, the pageEncoding attribute of the page directive of the JSP document, and in a JSP configuration element whose URL pattern matches the document.

A JSP container must raise a translation-time error if an unsupported page character encoding is requested.

JSP.4.2 Response Character Encoding

The response character encoding is the character encoding of the response generated from a JSP page, if that response is in the form of text. It is primarily managed as the javax.servlet.ServletResponse object's characterEncoding property.

The JSP container determines an initial response character encoding along with the initial content type for a JSP page and calls ServletResponse.setContentType() with this information before processing the page. JSP pages can set initial content type and initial response character encoding using the contentType attribute of the page directive.

The initial response content type is set to the TYPE value of the contentType attribute of the page directive. If the page doesn't provide this attribute, the initial content type is "text/html" for JSP pages in standard syntax and "text/xml" for JSP documents in XML syntax.

The initial response character encoding is set to the CHARSET value of the contentType attribute of the page directive. If the page doesn't provide this attribute or the attribute doesn't have a CHARSET value, the initial response character encoding is determined as follows:

- For documents in XML syntax, it is UTF-8.
- For JSP pages in standard syntax, it is the character encoding specified by the pageEncoding attribute of the page directive or by a JSP configuration element page-encoding whose URL pattern matches the page. Only the character encoding specified for the requested page is used; the encodings of files included via the include directive are not taken into consideration. If there's no such specification, no initial response character encoding is passed to ServletResponse.setContentType() - the ServletResponse object's default, ISO-8859-1, is used.

After the initial response character encoding has been set, the JSP page's content can dynamically modify it by calling the ServletResponse object's setCharacterEncoding and setLocale methods directly or indirectly. A number of JSTL internationalization and formatting actions call ServletResponse.setLocale(), which may affect the response character encoding. See the Servlet and JSTL specifications for more information.

Note that the response character encoding can only be changed until the response is committed. Data is sent to the response stream on buffer flushes for buffered pages, or on encountering the first content (beware of whitespace) on unbuffered pages. Whitespace is notoriously tricky for JSP Pages in JSP syntax, but much more manageable for JSP Documents in XML syntax.

JSP.4.3 Request Character Encoding

The request character encoding is the character encoding in which parameters in an incoming request are interpreted. It is primarily managed as the ServletRequest object's characterEncoding property.

The JSP specification doesn't provide functionality to handle the request character encoding directly. To control the request character encoding from JSP pages without embedded Java code, the JSTL <fmt:requestEncoding> can be used.

JSP.4.4 XML View Character Encoding

The XML view character encoding is the character encoding used for externalizing the XML view of a JSP page or tag file.

The XML view character encoding is always UTF-8.

JSP.4.5 Delivering Localized Content

The JSP specification does not mandate any specific approach for structuring localized content, and different approaches are possible. Two common approaches are to use a template taglib and pull localized strings from a resource repository, or to use-per-locale JSP pages. Each approach has benefits and drawbacks. The JSTL internationalization and formatting actions provide support for retrieving localized content from resource bundles and thus support the first approach. Some users have been using transformations on JSP documents to do simple replacement of elements

by localized strings, thus maintaining JSP syntax with no performance cost at run-time. Combinations of these approaches also make sense.

CHAPTER JSP.5

Standard Actions

This chapter describes the standard actions of JavaServer Pages 2.0 (JSP 2.0). Standard actions are represented using XML elements with a prefix of jsp (though that prefix can be redefined in the XML syntax). A translation error will result if the JSP prefix is used for an element that is not a standard action.

JSP.5.1 <jsp:useBean>

A jsp:useBean action associates an instance of a Java programming language object defined within a given scope and available with a given id with a newly declared scripting variable of the same id.

When a <jsp:useBean> action is used in an scriptless page, or in an scriptless context (as in the body of an action so indicated), there are no Java scripting variables created but instead an EL variable is created.

The jsp:useBean action is quite flexible; its exact semantics depends on the attributes given. The basic semantic tries to find an existing object using id and scope. If the object is not found it will attempt to create the object using the other attributes.

It is also possible to use this action to give a local name to an object defined elsewhere, as in another JSP page or in a servlet. This can be done by using the type attribute and not providing class or beanName attributes.

At least one of type and class must be present, and it is not valid to provide both class and beanName. If type and class are present, class must be assignable to type (in the Java platform sense). For it not to be assignable is a translation-time error.

The attribute beanName specifies the name of a Bean, as specified in the JavaBeans specification. It is used as an argument to the instantiate method in the java.beans.Beans class. It must be of the form a.b.c, which may be either a class,

or the name of a resource of the form a/b/c.ser that will be resolved in the current ClassLoader. If this is not true, a request-time exception, as indicated in the semantics of the instantiate method will be raised. The value of this attribute can be a request-time attribute expression.

The id Attribute

The id="name" attribute/value tuple in a jsp:useBean action has special meaning to a JSP container, at page translation time and at client request processing time. In particular:

- the name must be unique within the translation unit, and identifies the particular element in which it appears to the JSP container and page.

  Duplicate id's found in the same translation unit shall result in a fatal translation error.
- The JSP container will associate an object (a JavaBean component) with the named value and accessed via that name in various contexts through the pageContext object described later in this specification.

  The name is also used to expose a variable (name) in the page's scripting language environment. The scope of the scripting language variable is dependent upon the scoping rules and capabilities of the scripting language used in the page.
  Note that this implies the name value syntax must comply with the variable naming syntax rules of the scripting language used in the page. Chapter JSP.9 provides details for the case where the language attribute is java.

  An example of the scope rules just mentioned is shown next:

```
<% { // introduce a new block %>
    ...
    <jsp:useBean id="customer" class="com.myco.Customer" />

    <%
    /*
     * the tag above creates or obtains the Customer Bean
     * reference, associates it with the name "customer" in the
     * PageContext, and declares a Java programming language
     * variable of the same name initialized to the object reference
     * in this block's scope.
     */
    %>
    ...
    <%= customer.getName(); %>
    ...
<% } // close the block %>

<%
// the variable customer is out of scope now but
// the object is still valid (and accessible via pageContext)
%>
```

The scope Attribute

The scope="page|request|session|application" attribute/value tuple is associated with, and modifies the behavior of the id attribute described above (it has both translation time and client request processing time semantics). In particular it describes the namespace, the implicit lifecycle of the object reference associated with the name, and the APIs used to access this association. For all scopes, it is illegal to change the instance object so associated, such that its new runtime type is a subset of the type(s) of the object previously so associated. See Section JSP.1.8.2 for details on the available scopes.

Semantics

The actions performed in a jsp:useBean action are:

1. An attempt to locate an object based on the attribute values id and scope. The inspection is done synchronized per scope namespace to avoid non-deterministic behavior.
2. A scripting language variable of the specified type (if given) or class (if type

is not given) is defined with the given id in the current lexical scope of the scripting language. The type attribute should be used to specify a Java type that cannot be instantiated as a JavaBean (i.e. a Java type that is an abstract class, interface, or a class with no public no-args constructor). If the class attribute is used for a Java type that cannot be instantiated as a JavaBean, the container may consider the page invalid, and is recommended to (but not required to) produce a fatal translation error at translation time, or a java.lang.InstantiationException at request time. Similarly, if either type or class specify a type that can not be found, the container may consider the page invalid, and is recommended to (but not required to) produce a fatal translation error at translation time, or a java.lang.ClassNotFoundException at request time.

3. If the object is found, the variable's value is initialized with a reference to the located object, cast to the specified type. If the cast fails, a java.lang.ClassCastException shall occur. This completes the processing of this jsp:useBean action.

4. If the jsp:useBean action had a non-empty body it is ignored. This completes the processing of this jsp:useBean action.

5. If the object is not found in the specified scope and neither class nor beanName are given, a java.lang.InstantiationException shall occur. This completes the processing of this jsp:useBean action.

6. If the object is not found in the specified scope, and the class specified names a non-abstract class that defines a public no-args constructor, then the class is instantiated. The new object reference is associated with the scripting variable and with the specified name in the specified scope using the appropriate scope dependent association mechanism (see PageContext). After this, step 8 is performed.

   If the object is not found, and the class is either abstract, an interface, or no public no-args constructor is defined therein, then a java.lang.InstantiationException shall occur. This completes the processing of this jsp:useBean action.

7. If the object is not found in the specified scope; and beanName is given, then the method instantiate of java.beans.Beans will be invoked with the ClassLoader of the servlet object and the beanName as arguments. If the method succeeds, the new object reference is associated the with the scripting variable and with the specified name in the specified scope using the appropriate scope dependent association mechanism (see PageContext). After this, step 8 is performed.

8. If the jsp:useBean action has a non-empty body, the body is processed. The variable is initialized and available within the scope of the body. The text of the body is treated as elsewhere. Any template text will be passed through to

the out stream. Scriptlets and action tags will be evaluated.

A common use of a non-empty body is to complete initializing the created instance. In that case the body will likely contain jsp:setProperty actions and scriptlets that are evaluated. This completes the processing of this useBean action.

Examples

In the following example, a Bean with name connection of type com.myco.myapp.Connection is available after actions on this element, either because it was already created and found, or because it is newly created.

```
<jsp:useBean id=”connection” class=”com.myco.myapp.Connection” />
```

In the next example, the timeout property is set to 33 if the Bean was instantiated.

```
<jsp:useBean id=”connection” class=”com.myco.myapp.Connection”>
    <jsp:setProperty name=”connection” property=”timeout” value=”33”>
</jsp:useBean>
```

In the final example, the object should have been present in the session. If so, it is given the local name wombat with WombatType. A ClassCastException may be raised if the object is of the wrong class, and an InstantiationException may be raised if the object is not defined.

```
<jsp:useBean id=”wombat” type=”my.WombatType” scope=”session”/>
```

Syntax

This action may or not have a body. If the action has no body, it is of the form:

```
<jsp:useBean id="name" scope="page|request|session|application" typeSpec />

typeSpec ::= class=”className” |
    class=”className” type=”typeName” |
    type=”typeName” class=”className” |
    beanName=”beanName” type=”typeName” |
    type=”typeName” beanName=”beanName” |
    type=”typeName”
```

If the action has a body, it is of the form:

```
<jsp:useBean id="name" scope="page|request|session|application" typeSpec >
    body
</jsp:useBean>
```

In this case, the body will be invoked if the Bean denoted by the action is created. Typically, the body will contain either scriptlets or jsp:setProperty tags that will be used to modify the newly created object, but the contents of the body are not restricted.

The <jsp:useBean> tag has the following attributes:

Table JSP.5-1 *jsp:useBean Attributes*

id	The name used to identify the object instance in the specified scope's namespace, and also the scripting variable name declared and initialized with that object reference. The name specified is case sensitive and shall conform to the current scripting language variable-naming conventions.
scope	The scope within which the reference is available. The default value is page. See the description of the scope attribute defined earlier herein. A translation error must occur if scope is not one of "page", "request", "session" or "application".
class	The fully qualified name of the class that defines the implementation of the object. The class name is case sensitive. If the class and beanName attributes are not specified the object must be present in the given scope.
beanName	The name of a bean, as expected by the instantiate method of the java.beans.Beans class. This attribute can accept a request-time attribute expression as a value.

Table JSP.5-1 *jsp:useBean Attributes*

type	If specified, it defines the type of the scripting variable defined. This allows the type of the scripting variable to be distinct from, but related to, the type of the implementation class specified. The type is required to be either the class itself, a superclass of the class, or an interface implemented by the class specified. The object referenced is required to be of this type, otherwise a java.lang.ClassCastException shall occur at request time when the assignment of the object referenced to the scripting variable is attempted. If unspecified, the value is the same as the value of the class attribute.

JSP.5.2 <jsp:setProperty>

The jsp:setProperty action sets the values of properties in a bean. The name attribute that denotes the bean must be defined before this action appears.

There are two variants of the jsp:setProperty action. Both variants set the values of one or more properties in the bean based on the type of the properties. The usual bean introspection is done to discover what properties are present, and, for each, its name, whether it is simple or indexed, its type, and the setter and getter methods. Introspection also indicates if a given property type has a PropertyEditor class.

Properties in a Bean can be set from one or more parameters in the request object, from a String constant, or from a computed request-time expression. Simple and indexed properties can be set using jsp:setProperty.

When assigning from a parameter in the request object, the conversions described in Section JSP.1.14.2.1 are applied, using the target property to determine the target type.

When assigning from a value given as a String constant, the conversions described in Section JSP.1.14.2.1 are applied, using the target property to determine the target type.

When assigning from a value given as a request-time attribute, no type conversions are applied if a scripting expression is used, as indicated in Section JSP.1.14.2.2. If an EL expression is used, the type conversions described in Section JSP.2.8 are performed.

When assigning values to indexed properties the value must be an array; the rules described in the previous paragraph apply to the actions.

A conversion failure leads to an error, whether at translation time or request-time.

Examples

The following two actions set a value from the request parameter values.

```
<jsp:setProperty name=”request” property=”*” />
<jsp:setProperty name=”user” property=”user” param=”username” />
```

The following two elemenst set a property from a value

```
<jsp:setProperty name=”results” property=”col” value=”${i mod 4}”/>
<jsp:setProperty name=”results” property=”row” value=”<%= i/4 %>” />
```

Syntax

```
<jsp:setProperty name="beanName" prop_expr />

prop_expr ::=
        property="*"       |
        property=”propertyName”|
        property=”propertyName” param="parameterName"|
        property=”propertyName” value=”propertyValue”
```

propertyValue ::= *string*

The value propertyValue can also be a request-time attribute value, as described in Section JSP.1.14.1.

propertyValue ::= expr_scriptlet[1]

[1.] See syntax for expression scriptlet <%= ... %>

The <jsp:setProperty> action has the following attributes:

Table JSP.5-2 *jsp:setProperty Attributes*

name	The name of a bean instance defined by a <jsp:useBean> action or some other action. The bean instance must contain the property to be set. The defining action must appear before the <jsp:setProperty> action in the same file.
property	The name of the property whose value will be set. If propertyName is set to * then the tag will iterate over the current ServletRequest parameters, matching parameter names and value type(s) to property names and setter method type(s), setting each matched property to the value of the matching parameter. If a parameter has a value of "", the corresponding property is not modified.
param	The name of the request parameter whose value is given to a bean property. The name of the request parameter usually comes from a web form. If param is omitted, the request parameter name is assumed to be the same as the bean property name. If the param is not set in the Request object, or if it has the value of "", the jsp:setProperty action has no effect (a noop). An action may not have both param and value attributes.
value	The value to assign to the given property. This attribute can accept a request-time attribute expression as a value. An action may not have both param and value attributes.

JSP.5.3 <jsp:getProperty>

The <jsp:getProperty> action places the value of a bean instance property, converted to a String, into the implicit out object, from which the value can be displayed as output. The bean instance must be defined as indicated in the name attribute before this point in the page (usually via a jsp:useBean action).

The conversion to String is done as in the println methods, i.e. the toString method of the object is used for Object instances, and the primitive types are converted directly.

If the object is not found, a request-time exception is raised.

The value of the name attribute in jsp:setProperty and jsp:getProperty will refer to an object that is obtained from the pageContext object through its findAttribute method.

The object named by the name must have been “introduced” to the JSP processor using either the jsp:useBean action or a custom action with an associated VariableInfo entry for this name. If the object was not introduced in this manner, the container implementation is recommended (but not required) to raise a translation error, since the page implementation is in violation of the specification.

Note – A consequence of the previous paragraph is that objects that are stored in, say, the session by a front component are not automatically visible to jsp:setProperty and jsp:getProperty actions in that page unless a jsp:useBean action, or some other action, makes them visible.

If the JSP processor can ascertain that there is an alternate way guaranteed to access the same object, it can use that information. For example it may use a scripting variable, but it must guarantee that no intervening code has invalidated the copy held by the scripting variable. The truth is always the value held by the pageContext object.

Examples

```
<jsp:getProperty name=”user” property=”name” />
```

Syntax

```
<jsp:getProperty name=”name” property=”propertyName” />
```

The attributes are:

Table JSP.5-3 *jsp:getProperty Attributes*

name	The name of the object instance from which the property is obtained.
property	Names the property to get.

JSP.5.4 <jsp:include>

A <jsp:include .../> action provides for the inclusion of static and dynamic resources in the same context as the current page. See Table JSP.1-10 for a summary of include facilities.

Inclusion is into the current value of out. The resource is specified using a relativeURLspec that is interpreted in the context of the web application (i.e. it is mapped).

The page attribute of both the jsp:include and the jsp:forward actions are interpreted relative to the current JSP page, while the file attribute in an include directive is interpreted relative to the current JSP file. See below for some examples of combinations of this.

An included page cannot change the response status code or set headers. This precludes invoking methods like setCookie. Attempts to invoke these methods will be ignored. The constraint is equivalent to the one imposed on the include method of the RequestDispatcher class.

A jsp:include action may have jsp:param subelements that can provide values for some parameters in the request to be used for the inclusion.

Request processing resumes in the calling JSP page, once the inclusion is completed.

The flush attribute controls flushing. If true, then, if the page output is buffered and the flush attribute is given a true value, then the buffer is flushed prior to the inclusion, otherwise the buffer is not flushed. The default value for the flush attribute is false.

Examples

```
<jsp:include page="/templates/copyright.html"/>
```

The above example is a simple inclusion of an object. The path is interpreted in the context of the Web Application. It is likely a static object, but it could be mapped into, for instance, a servlet via web.xml.

For an example of a more complex set of inclusions, consider the following four situations built using four JSP files: A.jsp, C.jsp, dir/B.jsp and dir/C.jsp:

- A.jsp says <%@ include file="dir/B.jsp"%> and dir/B.jsp says <%@ include file="C.jsp"%>. In this case the relative specification C.jsp resolves to dir/C.jsp.
- A.jsp says <jsp:include page="dir/B.jsp"/> and dir/B.jsp says <jsp:include page="C.jsp" />. In this case the relative specification C.jsp resolves to dir/C.jsp.
- A.jsp says <jsp:include page="dir/B.jsp"/> and dir/B.jsp says <%@ include file="C.jsp" %>. In this case the relative specification C.jsp resolves to dir/C.jsp.
- A.jsp says <%@ include file="dir/B.jsp"%> and dir/B.jsp says <jsp:include page="C.jsp"/>. In this case the relative specification C.jsp resolves to C.jsp.

Syntax

```
<jsp:include page="urlSpec" flush="true|false"/>
```

and

```
<jsp:include page="urlSpec" flush="true|false">
    { <jsp:param .... /> }*
</jsp:include>
```

The first syntax just does a request-time inclusion. In the second case, the values in the param subelements are used to augment the request for the purposes of the inclusion.

The valid attributes are:

Table JSP.5-4 *jsp:include Atrributes*

page	The URL is a relative urlSpec as in Section JSP.1.2.1. Relative paths are interpreted relative to the current JSP page. Accepts a request-time attribute value (which must evaluate to a String that is a relative URL specification).
flush	Optional boolean attribute. If the value is true, the buffer is flushed now. The default value is false.

JSP.5.5 <jsp:forward>

A <jsp:forward page="urlSpec" /> action allows the runtime dispatch of the current request to a static resource, a JSP page or a Java servlet class in the same con-

text as the current page. A jsp:forward effectively terminates the execution of the current page. The relative urlSpec is as in Section JSP.1.2.1.

The request object will be adjusted according to the value of the `page` attribute.

A jsp:forward action may have jsp:param subelements that can provide values for some parameters in the request to be used for the forwarding.

If the page output is buffered, the buffer is cleared prior to forwarding.

If the page output is buffered and the buffer was flushed, an attempt to forward the request will result in an IllegalStateException.

If the page output was unbuffered and anything has been written to it, an attempt to forward the request will result in an IllegalStateException.

Examples

The following action might be used to forward to a static page based on some dynamic condition.

```
<% String whereTo = “/templates/”+someValue; %>
<jsp:forward page=’<%= whereTo %>’ />
```

Syntax

```
<jsp:forward page=”relativeURLspec” />
```

and

```
<jsp:forward page=”urlSpec”>
    { <jsp:param .... /> }*
</jsp:forward>
```

This tag allows the page author to cause the current request processing to be affected by the specified attributes as follows:

Table JSP.5-5 *jsp:forward Attributes*

page	The URL is a relative urlSpec as in Section JSP.1.2.1. Relative paths are interpreted relative to the current JSP page. Accepts a request-time attribute value (which must evaluate to a String that is a relative URL specification).

JSP.5.6 <jsp:param>

The jsp:param element is used to provide key/value information. This element is used in the jsp:include, jsp:forward, and jsp:params elements. A translation error shall occur if the element is used elsewhere.

When doing jsp:include or jsp:forward, the included page or forwarded page will see the original request object, with the original parameters augmented with the new parameters, with new values taking precedence over existing values when applicable. The scope of the new parameters is the jsp:include or jsp:forward call; i.e. in the case of an jsp:include the new parameters (and values) will not apply after the include. This is the same behavior as in the ServletRequest include and forward methods (see Section 8.1.1 in the Servlet 2.4 specification).

For example, if the request has a parameter A=foo and a parameter A=bar is specified for forward, the forwarded request shall have A=bar,foo. Note that the new param has precedence.

The parameter names and values specified should be left unencoded by the page author. The JSP container must encode the parameter names and values using the character encoding from the request object when necessary. For example, if the container chooses to append the parameters to the URL in the dispatched request, both the names and values must be encoded as per the content type application/x-www-form-urlencoded in the HTML specification.

Syntax

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

This action has two mandatory attributes: name and value. name indicates the name of the parameter, and value, which may be a request-time expression, indicates its value.

JSP.5.7 <jsp:plugin>

The plugin action enables a JSP page author to generate HTML that contains the appropriate client browser dependent constructs (OBJECT or EMBED) that will result in the download of the Java Plugin software (if required) and subsequent execution of the Applet or JavaBeans component specified therein.

The <jsp:plugin> tag is replaced by either an <object> or <embed> tag, as appropriate for the requesting user agent, and emitted into the output stream of the

response. The attributes of the <jsp:plugin> tag provide configuration data for the presentation of the element, as indicated in the table below.

The <jsp:params> action containing one or more <jsp:param> actions provides parameters to the Applet or JavaBeans component.

The <jsp:fallback> element indicates the content to be used by the client browser if the plugin cannot be started (either because OBJECT or EMBED is not supported by the client browser or due to some other problem). If the plugin can start but the Applet or JavaBeans component cannot be found or started, a plugin specific message will be presented to the user, most likely a popup window reporting a ClassNotFoundException.

The actual plugin code need not be bundled with the JSP container and a reference to Sun's plugin location can be used instead, although some vendors will choose to include the plugin for the benefit of their customers.

Examples

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

Syntax

```
<jsp:plugin type="bean|applet"
    code="objectCode"
    codebase="objectCodebase"
    { align="alignment"          }
    { archive="archiveList"      }
    { height="height"            }
    { hspace="hspace"            }
    { jreversion="jreversion"    }
    { name="componentName"       }
    { vspace="vspace"            }
    { title="title"              }
    { width="width"              }
    { nspluginurl="url"          }
    { iepluginurl="url"          }
    { mayscript='true|false'     } >

    { <jsp:params>
        { <jsp:param name="paramName" value="paramValue" /> }+
     </jsp:params> }

    { <jsp:fallback> arbitrary_text </jsp:fallback> }
</jsp:plugin>
```

Table JSP.5-6 *jsp:plugin Attributes*

type	Identifies the type of the component; a bean, or an Applet.
code	As defined by HTML spec
codebase	As defined by HTML spec
align	As defined by HTML spec
archive	As defined by HTML spec
height	As defined by HTML spec. Accepts a run-time expression value.
hspace	As defined by HTML spec.
jreversion	Identifies the spec version number of the JRE the component requires in order to operate; the default is: 1.2
name	As defined by HTML spec

Table JSP.5-6 *jsp:plugin Attributes*

vspace	As defined by HTML spec
title	As defined by the HTML spec
width	As defined by HTML spec. Accepts a run-time expression value.
nspluginurl	URL where JRE plugin can be downloaded for Netscape Navigator, default is implementation defined.
iepluginurl	URL where JRE plugin can be downloaded for IE, default is implementation defined.
mayscript	As defined by HTML spec.

JSP.5.8 <jsp:params>

The jsp:params action is part of the jsp:plugin action and can only occur as a direct child of a <jsp:plugin> action. Using the jsp:params element in any other context shall result in a translation-time error.

The semantics and syntax of jsp:params are described in Section JSP.5.7.

JSP.5.9 <jsp:fallback>

The jsp:fallback action is part of the jsp:plugin action and can only occur as a direct child of a <jsp:plugin> element. Using the jsp:fallback element in any other context shall result in a translation-time error.

The semantics and syntax of jsp:fallback are described in Section JSP.5.7.

JSP.5.10 <jsp:attribute>

The <jsp:attribute> standard action has two uses. It allows the page author to define the value of an action attribute in the body of an XML element instead of in the value of an XML attribute. It also allows the page author to specify the attributes of the element being output, when used inside a <jsp:element> action. The action must only appear as a subelement of a standard or custom action. An attempt to use it otherwise must result in a translation error. For example, it cannot be used to specify the value of an attribute for XML elements that are template

text. For custom action invocations, JSP containers must support the use of <jsp:attribute> for both Classic and Simple Tag Handlers.

The behavior of the <jsp:attribute> standard action varies depending on the type of attribute being specified, as follows:

- A translation error must occur if <jsp:attribute> is used to define the value of an attribute of <jsp:attribute>.
- If the enclosing action is <jsp:element>, the value of the name attribute and the body of the action will be used as attribute name/value pairs in the dynamically constructed element. See Section JSP.5.14 for more details on <jsp:element>. Note that in this context, the attribute does not apply to the <jsp:element> action itself, but rather to the output of the element. That is, <jsp:attribute> cannot be used to specify the name attribute of the <jsp:element> action.
- For custom action attributes of type javax.servlet.jsp.tagext.JspFragment, the container must create a JspFragment out of the body of the <jsp:attribute> action and pass it to the tag handler. This applies for both Classic Tag Handlers and Simple Tag Handlers. A translation error must result if the body of the <jsp:attribute> action is not scriptless in this case.
- If the custom action accepts dynamic attributes (Section JSP.7.1.8), and the name of the attribute is not one explicitly indicated for the tag, then the container will evaluate the body of <jsp:attribute> and assign the computed value to the attribute using the dynamic attribute machinery. Since the type of the attribute is unknown and the body of <jsp:attribute> evaluates to a String, the container must pass in an instance of String.
- For standard or custom action attributes that accept a request-time expression value, the Container must evaluate the body of the <jsp:attribute> action and use the result of this evaluation as the value of the attribute. The body of the attribute action can be any JSP content in this case. If the type of the attribute is not String, the standard type conversion rules are applied, as per Section JSP.1.14.2.1, “Conversions from String values”.
- For standard or custom action attributes that do not accept a request-time expression value, the Container must use the body of the <jsp:attribute> action as the value of the attribute. A translation error must result if the body of the <jsp:attribute> action contains anything but template text.

If the body of the <jsp:attribute> action is empty, it is the equivalent of specifying “” as the value of the attribute. Note that after being trimmed, non-empty bodies can result in a value of “” as well.

The <jsp:attribute> action accepts a name attribute and a trim attribute. The name attribute associates the action with one of the attributes the tag handler is declared to accept, or in the case of <jsp:element> it associates the action with one of the attributes in the element being output. The optional trim attribute determines whether the whitespace appearning at the beginning and at the end of the element body should be discarded or not. By default, the leading and trailing whitespace is discarded. The Container must trim at translation time only. The Container must not trim at runtime. For example, if a body contains a custom action that produces leading or trailing whitespace, that whitespace is preserved regardless of the value of the trim attribute.

The following is an example of using the <jsp:attribute> standard action to define an attribute that is evaluated by the container prior to the custom action invocation. This example assumes the name attribute is declared with type java.lang.String in the TLD.

```
<mytag:highlight>
    <jsp:attribute name=”text”>
        Inline definition.
    </jsp:attribute>
</mytag:highlight>
```

The following is an example of using the <jsp:attribute> standard action within <jsp:element>, to define which attributes are to be output with that element:

```
<jsp:element name=”firstname”>
    <jsp:attribute name=”name”>Susan</jsp:attribute>
</jsp:element>
```

This would produce the following output:

```
<firstname name=”Susan”/>
```

See Section JSP.1.3.10, “JSP Syntax Grammar” for the formal syntax definition of the <jsp:attribute> standard action.

The attributes are:

Table JSP.5-7 *Attributes for the <jsp:attribute> standard action*

name	(required) If not being used with <jsp:element>, then if the action does not accept dynamic attributes, the name must match the name of an attribute for the action being invoked, as declared in the Tag Library Descriptor for a custom action, or as specified for a standard action, or a translation error will result. Except for when used with <jsp:element>, a translation error will result if both an XML element attribute and a <jsp:attribute> element are used to specify the value for the same attribute. The value of name can be a QName. If so, a translation error must occur if the prefix does not match that of the action it applies to, unless the action supports dynamic attributes, or unless the action is <jsp:element>. When used with <jsp:element>, this attribute specifies the name of the attribute to be included in the generated element.
trim	(optional) Valid values are true and false. If true, the whitespace, including spaces, carriage returns, line feeds, and tabs, that appears at the beginning and at the end of the body of the <jsp:attribute> action will be ignored by the JSP compiler. If false the whitespace is not ignored. Defaults to true.

JSP.5.11 <jsp:body>

Normally, the body of a standard or custom action invocation is defined implicitly as the body of the XML element used to represent the invocation. The body of a standard or custom action can also be defined explicitly using the <jsp:body> standard action. This is required if one or more <jsp:attribute> elements appear in the body of the tag.

If one or more <jsp:attribute> elements appear in the body of a tag invocation but no <jsp:body> element appears or an empty <jsp:body> element appears, it is the equivalent of the tag having an empty body.

It is also legal to use the <jsp:body> standard action to supply bodies to standard actions, for any standard action that accepts a body (except for

<jsp:body>, <jsp:attribute>, <jsp:scriptlet>, <jsp:expression>, and <jsp:declaration>).

The body standard action accepts no attributes.

JSP.5.12 <jsp:invoke>

The <jsp:invoke> standard action can only be used in tag files (see Chapter JSP.8, “Tag Files”), and must result in a translation error if used in a JSP. It takes the name of an attribute that is a fragment, and invokes the fragment, sending the output of the result to the JspWriter, or to a scoped attribute that can be examined and manipulated. If the fragment identified by the given name is null, <jsp:invoke> will behave as though a fragment was passed in that produces no output.

JSP.5.12.1 Basic Usage

The most basic usage of this standard action will invoke a fragment with the given name with no parameters. The fragment will be invoked using the JspFragment.invoke method, passing in null for the Writer parameter so that the results will be sent to the JspWriter of the JspContext associated with the JspFragment. The following is an example of such a basic fragment invocation:

```
<jsp:invoke fragment=”frag1”/>
```

JSP.5.12.2 Storing Fragment Output

It is also possible to invoke the fragment and send the results to a scoped attribute for further examination and manipulation. This can be accomplished by specifying the var or varReader attribute in the action. In this usage, the fragment is invoked using the JspFragment.invoke method, but a custom java.io.Writer is passed in instead of null.

If var is specified, the container must ensure that a java.lang.String object is made available in a scoped attribute with the name specified by var. The String must contain the content sent by the fragment to the Writer provided in the JspFragment.invoke call.

If varReader is specified, the container must ensure that a java.io.Reader object is constructed and is made available in a scoped attribute with the name specified by varReader. The Reader object can then be passed to a custom action for further processing. The Reader object must produce the content sent by the

fragment to the provided Writer. The Reader must also be resettable. That is, if its reset method is called, the result of the invoked fragment must be able to be read again without re-executing the fragment.

An optional scope attribute indicates the scope of the resulting scoped variable.

The following is an example of using var or varReader and the scope attribute:

```
<jsp:invoke fragment="frag2" var="resultString" scope="session"/>

<jsp:invoke fragment="frag3" varReader="resultReader" scope="page"/>
```

JSP.5.12.3 Providing a Fragment Access to Variables

JSP fragments have access to the same page scope variables as the page or tag file in which they were defined (in addition to variables in the request, session, and application scopes). Tag files have access to a local page scope, separate from the page scope of the calling page. When a tag file invokes a fragment that appears in the calling page, the JSP container provides a way to synchronize variables between the local page scope in the tag file and the page scope of the calling page. For each variable that is to be synchronized, the tag file author must declare the variable with a scope of either AT_BEGIN or NESTED. The container must then generate code to synchronize the page scope values for the variable in the tag file with the page scope equivalent in the calling page or tag file. The details of how variables are synchronized can be found in Section JSP.8.9.

The following is an example of a tag file providing a fragment access to a variable:

```
<%@ variable name-given="x" scope="NESTED" %>
...
<c:set var="x" value="1"/>
<jsp:invoke fragment="frag4"/>
```

A translation error shall result if the <jsp:invoke> action contains a non-empty body.

See Section JSP.1.3.10, "JSP Syntax Grammar" for the formal syntax definition of the <jsp:invoke> standard action.

The attributes are:

Table JSP.5-8 *Attributes for the <jsp:invoke> standard action*

fragment	(required) The name used to identify this fragment during this tag invocation.
var	(optional) The name of a scoped attribute to store the result of the fragment invocation in, as a java.lang.String object. A translation error must occur if both var and varReader are specified. If neither var nor varReader are specified, the result of the fragment goes directly to the JspWriter, as described above.
varReader	(optional) The name of a scoped attribute to store the result of the fragment invocation in, as a java.io.Reader object. A translation error must occur if both var and varReader are specified. If neither var nor varReader is specified, the result of the fragment invocation goes directly to the JspWriter, as described above.
scope	(optional) The scope in which to store the resulting variable. A translation error must result if the value is not one of page, request, session, or application. A translation error will result if this attribute appears without specifying either the var or varReader attribute as well. Note that a value of session should be used with caution since not all calling pages may be participating in a session. A container must throw an IllegalStateException at runtime if scope is session and the calling page does not participate in a session. Defaults to page.

JSP.5.13 <jsp:doBody>

The <jsp:doBody> standard action can only be used in tag files (see Chapter JSP.8, “Tag Files”), and must result in a translation error if used in a JSP. It invokes the body of the tag, sending the output of the result to the JspWriter, or to a scoped attribute that can be examined and manipulated.

The <jsp:doBody> standard action behaves exactly like <jsp:invoke>, except that it operates on the body of the tag instead of on a specific fragment passed as an attribute. Because it always operates on the body of the tag, there is no name attribute for this standard action. The var, varReader, and scope attributes are all

supported with the same semantics as for <jsp:invoke>. Fragments are provided access to variables the same way for <jsp:doBody> as they are for <jsp:invoke>. If no body was passed to the tag, <jsp:doBody> will behave as though a body was passed in that produces no output.

The body of a tag is passed to the simple tag handler as a JspFragment object.

A translation error shall result if the <jsp:doBody> action contains a non-empty body.

See Section JSP.1.3.10, "JSP Syntax Grammar" for the formal syntax definition of the <jsp:doBody> standard action.

The attributes are:

Table JSP.5-9 *Attributes for the <jsp:doBody> standard action*

var	(optional) The name of a scoped attribute to store the result of the body invocation in, as a java.lang.String object. A translation error must occur if both var and varReader are specified. If neither var nor varReader are specified, the result of the body goes directly to the JspWriter, as described above.
varReader	(optional) The name of a scoped attribute to store the result of the body invocation in, as a java.io.Reader object. A translation error must occur if both var and varReader are specified. If neither var nor varReader is specified, the result of the body invocation goes directly to the JspWriter, as described above.
scope	(optional) The scope in which to store the resulting variable. A translation error must result if the value is not one of page, request, session, or application. A translation error will result if this attribute appears without specifying either the var or varReader attribute as well. Note that a value of session should be used with caution since not all calling pages may be participating in a session. A container must throw an IllegalStateException at runtime if scope is session and the calling page does not participate in a session. Defaults to page.

JSP.5.14 <jsp:element>

The jsp:element action is used to dynamically define the value of the tag of an XML element. This action can be used in JSP pages, tag files and JSP documents.

This action has an optional body; the body can use the jsp:attribute and jsp:body actions.

A jsp:element action has one mandatory attribute, name, of type String. The value of the attribute is used as that of the tag of the element generated.

Examples

The following example generates an XML element whose name depends on the result of an EL expression, *content.headerName*. The element has an attribute, *lang*, and the value of the attribute is that of the expression *content.lang*. The body of the element is the value of the expression *content.body.*

```
<jsp:element
        name=”${content.headerName}”
        xmlns:jsp=”http://java.sun.com/JSP/Page”>
    <jsp:attribute name=”lang”>${content.lang}</jsp:attribute>
    <jsp:body>${content.body}</jsp:body>
</jsp:element>
```

The next example fragment shows that jsp:element needs no children. The example generates an empty element with name that of the value of the expression *myName*.

```
<jsp:element name=”${myName}”/>
```

Syntax

The jsp:element action may have a body. Two forms are valid, depending on whether the element is to have attributes or not. In the first form, no attributes are present:

```
<jsp:element name="name">
    optional body
</jsp:element>
```

In the second form, zero or more attributes are requested, using jsp:attribute and jsp:body, as appropriate.

```
<jsp:element name="name">
    jsp:attribute*
    jsp:body?
</jsp:element>
```

The one valid, mandatory, attribute of jsp:element is its name. Unlike other standard actions, the value of the name attribute must be given as an XML-style attribute and cannot be specified using <jsp:attribute> This is because <jsp:attribute> has a special meaning when used in the body of <jsp:element>. See Section JSP.5.10 for more details..

Table JSP.5-10 *Attributes for the <jsp:element> standard action*

name	(required) The value of name is that of the element genreated. The name can be a QName; JSP 2.0 places no constraints on this value: it is accepted as is. A request-time attribute value may be used for this attribute.

JSP.5.15 <jsp:text>

A jsp:text action can be used to enclose template data in a JSP page, a JSP document, or a tag file. A jsp:text action has no attributes and can appear anywhere that template data can. Its syntax is:

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

The interpretation of a jsp:text element is to pass its content through to the current value of out. This is very similar to the XSLT xsl:text element.

Examples

The following example is a fragment that could be in both a JSP page or a JSP document.

```
<jsp:text>
    This is some content
</jsp:text>
```

Expressions may appear within jsp:text, as in the next example, where the expression *foo.content* is evaluated and the result is inserted.

```
<jsp:text>
    This is some content: ${foo.content}
</jsp:text>
```

No subelements may appear within jsp:text; for example the following frag-

ment is invalid and must generate a translation error.

```
<jsp:text>
    This is some content: <jsp:text>foo</jsp:text>
</jsp:text>
```

When within a JSP document, of course, the body content needs to additionally conform to the constraints of being a well-formed XML document, so the following example, although valid in a JSP page is invalid in a JSP document:

```
<jsp:text>
    This is some content: ${foo.content > 3}
</jsp:text>
```

The same example can be made legal, with no semantic changes, by using *gt* instead of > in the expression; i.e. *${foo.content gt 3}*.

In an JSP document, CDATA sections can also be used to quote, uninterpreted, content, as in the following example:

```
<jsp:text>
    <![CDATA[<mumble></foobar>]]>
</jsp:text>
```

Syntax

The jsp:text action has no attributes. The action may have a body. The body may not have nested actions nor scripting elements. The body may have EL expressions. The syntax is of the form:

```
<jsp:text>
    optional body
</jsp:text>
```

JSP.5.16 <jsp:output>

The jsp:output action can only be used in JSP documents and in tag files in XML syntax, and a translation error must result if used in a standard syntax JSP or tag file. This action is used to modify some properties of the output of a JSP document or a tag file. In JSP 2.0 there are four properties that can be specified, all of which affect the output of the XML prolog.

The omit-xml-declaration property allows the page author to adjust whether an XML declaration is to be inserted at the beginning of the output. Since XML declarations only make sense for when the generated content is XML, the default value of this property is defined so that it is unnecessary in most cases.

The omit-xml-declaration property is of type String and the valid values are "yes", "no", "true" and "false". The name, values and semantics mimic that of the xsl:output element in the XSLT specification: if a value of "yes" or "true" is given, the container will not add an XML declaration; if a value of "no" or "false" is given, the container will add an XML declaration.

The default value for a JSP document that has a jsp:root element is "yes". The default value for JSP documents without a jsp:root element is "no".

The default value for a tag file in XML syntax is always "yes". If the value is "false" or "no" the tag file will emit an XML declaration as its first content.

The generated XML declaration is of the form:

<?xml version="1.0" encoding="*encodingValue*" ?>

Where *encodingValue* is the response character encoding, as determined in Section JSP.4.2 .

The doctype-root-element, doctype-system and doctype-public properties allow the page author to specify that a DOCTYPE be automatically generated in the XML prolog of the output. Without these properties, the DOCTYPE would need to be output manually via a <jsp:text> element before the root element of the JSP document, which is inconvenient.

A DOCTYPE must be automatically output if and only if the doctype-system element appears in the translation unit as part of a <jsp:output> action. The doctype-root-element must appear and must only appear if the doctype-system property appears, or a translation error must occur. The doctype-public property is optional, but must not appear unless the doctype-system property appears, or a translation error must occur.

The DOCTYPE to be automatically output, if any, is statically determined at translation time. Multiple occurrences of the doctype-root-element, doctype-system or doctype-public properties will cause a translation error if the values for the properties differ from the previous occurrence.

The DOCTYPE that is automatically output, if any, must appear immediately before the first element of the output document. The name following <!DOCTYPE must be the value of the doctype-root-element property. If a doctype-public property appears, then the format of the generated DOCTYPE is:

<!DOCTYPE nameOfRootElement PUBLIC “doctypePublic” “doctypeSystem”>

If a doctype-public property does not appear, then the format of the generated DOCTYPE is:

<!DOCTYPE nameOfRootElement SYSTEM “doctypeSystem”>

Where nameOfRootElement is the value of the doctype-root-element property, doctypePublic is the value of the doctype-public attribute, and doctypeSystem is the value of the doctype-system property.

The values for doctypePublic and doctypeSystem must be enclosed in either single or double quotes, depending on the value provided by the page author. It is the responsibility of the page author to provide a syntactically-valid URI as per the XML specification (see http://www.w3.org/TR/REC-xml#dt-sysid).

Examples

The following JSP document (with an extension of .jspx or with <is-xml> set to true in the JSP configuration):

```
<?xml version=”1.0” encoding=”EUC-JP” ?>
<hello></hello>
```

generates an XML document as follows:

```
<?xml version=”1.0” encoding=”UTF-8” ?>
<hello></hello>
```

The following JSP document is like the previous one, except that the XML declaration is omited. A typical use would be where the XML fragment is to be included within another document.

```
<?xml version=”1.0” encoding=”EUC-JP” ?>
<hello>
    <jsp:output
        xmlns:jsp=”http://java.sun.com/JSP/Page”
        omit-xml-declaration=”true”/>
</hello>
```

The following JSP document is equivalent but uses jsp:root instead of jsp:output.

```
<?xml version="1.0" encoding="EUC-JP" ?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
    <hello></hello>
</jsp:root>
```

The following JSP document specifies both a doctype-public and a doctype-system:

```
<?xml version="1.0" encoding="UTF-8" ?>
<html xmlns:jsp="http://java.sun.com/JSP/Page">
    <jsp:output doctype-root-element="html"
doctype-public="-//W3C//DTD XHTML Basic 1.0//EN"
doctype-system="http://www.w3.org/TR/xhtml-basic/xhtml-basic10.dtd" />
    <body>
        <h1>Example XHTML Document</h1>
    </body>
</html>
```

and generates and XML document as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML Basic 1.0//EN"
"http://www.w3.org/TR/xhtml-basic/xhtml-basic10.dtd">
<html><body><h1>Example XHTML Document</h1></body></html>
```

The following JSP document omits the doctype-public and explicitly omits the XML declaration:

```
<?xml version="1.0" encoding="UTF-8" ?>
<elementA>
    <jsp:output  omit-xml-declaration="true"
                 doctype-root-element="elementA"
                 doctype-system="test.dtd" />
    Element body goes here.
</elementA>
```

and generates an XML document as follows:

```
<!DOCTYPE elementA SYSTEM "test.dtd">
<elementA>Element body goes here.</elementA>
```

Syntax

The jsp:output action cannot have a body. The <jsp:output> action has the following syntax:

```
<jsp:output ( omit-xml-declaration="yes|no|true|false" ) { doctypeDecl } />

doctypeDecl ::=     (   doctype-root-element="rootElement"
                        doctype-public="PubidLiteral"
                        doctype-system="SystemLiteral" )
                  |  (  doctype-root-element="rootElement"
                        doctype-system="SystemLiteral" )
```

The following are the valid attributes of jsp:output:

Table JSP.5-11 *Attribute for the <jsp:output> standard action*

omit-xml-declaration	(optional) Indicates whether to omit the generation of an XML declaration. Acceptable values are “true”, “yes”, “false” and “no”.
doctype-root-element	(optional) Must be specified if and only if doctype-system is specified or a translation error must occur. Indicates the name that is to be output in the generated DOCTYPE declaration.
doctype-system	(optional) Specifies that a DOCTYPE declaration is to be generated and gives the value for the System Literal.
doctype-public	(optional) Must not be specified unless doctype-system is specified. Gives the value for the Public ID for the generated DOCTYPE.

JSP.5.17 Other Standard Actions

Chapter JSP.6 defines several other standard actions that are either convenient or needed to describe JSP pages with an XML document, some of which are available in all JSP pages. They are:

- <jsp:root>, defined in Section JSP.6.3.2.
- <jsp:declaration>, defined in Section JSP.6.3.7.

- <jsp:scriptlet>, defined in Section JSP.6.3.7.
- <jsp:expression>, defined in Section JSP.6.3.7.

CHAPTER JSP.6

JSP Documents

This chapter introduces two concepts related to XML and JSP: JSP documents and XML views. This chapter provides a brief overview of the two concepts and their relationship and also provides the details of JSP documents. The details of the XML view of a JSP document are described in Chapter JSP.10.

JSP.6.1 Overview of JSP Documents and of XML Views

A *JSP document* is a JSP page written using XML syntax. JSP documents need to be described as such, either implicitly or explicitly, to the JSP container, which will then process them as XML documents, checking for well-formedness and applying requests like entity declarations, if present. JSP documents are used to generate dynamic content using the standard JSP semantics.

Here is a simple JSP document:

```
<table>
    <c:forEach
        xmlns:c="http://java.sun.com/jsp/jstl/core"
        var="counter" begin="1" end="3">
            <row>${counter}</row>
    </c:forEach>
</table>
```

This well-formed, namespace-aware XML document generates, using the JSP standard tag library, an XML document that has <table> as the root element. That element has 3 <row> subelements containing values 1, 2 and 3. See Section JSP.6.4 for more details of this and other examples.

The design of JSP documents is focused on the generation of dynamic XML content, in any of its many uses, but JSP documents can be used to generate any dynamic content.

Some of the syntactic elements described in Chapter 1 are not legal XML; this chapter describes alternative syntaxes for those elements that are aligned with the XML syntax. The alternative syntaxes can be used in JSP documents; most of them (jsp:output and jsp:root are exceptions) can also be used in JSP pages in JSP syntax. As it will be described later, the alternative syntax is also used in the XML view of JSP pages.

JSP documents can be used in a number of ways, including:

- JSP documents can be passed directly to the JSP container; this is becoming more important as more and more content is authored as XML, be it in an XML-based languages like XHTML or SVG, or for the exchange of documents in applications like Web Services. The generated content may be sent directly to a client, or it may be part of some XML processing pipeline.
- JSP documents can be manipulated by XML-aware tools.
- A JSP document can be generated from a textual representation by applying an XML transformation, like XSLT.
- A JSP document can be generated automatically, say by serializing some objects

Tag files can also be authored using XML syntax. The rules are very similar to that of JSP documents; see Section JSP.8.6 for more details.

The *XML view of a JSP page* is an XML document that is derived from the JSP page following a mapping defined later in this chapter.. The XML view of a JSP page is intended to be used for validating the JSP page against some description of the set of valid pages. Validation of the JSP page is supported in the JSP 2.0 specification through a TagLibraryValidator class associated with a tag library. The validator class acts on a PageData object that represents the XML view of the JSP page (see, for example, Section JSP.7.4.1.2)

Figure JSP.6-1 below depicts the relationship between the concepts of JSP pages, JSP documents and XML views. Two phases are involved: the Translation phase, where JSP pages, in either syntax, are exposed to Tag Library Validators, via their XML view, and the Request Processing phase, where requests are processed to produce responses.

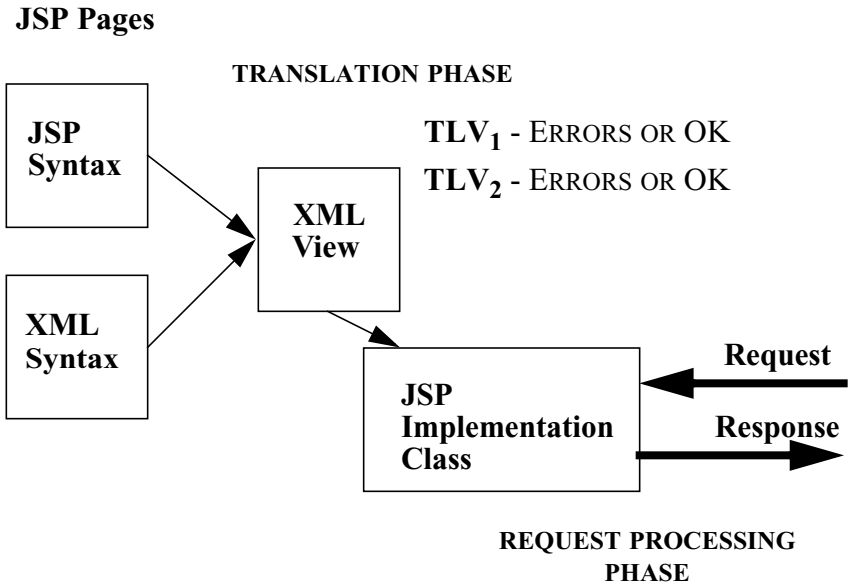


Figure JSP.6-1 Relationship between JSP Pages and XML views of JSP pages.

JSP documents are used by JSP page authors. They can be authored directly, using a text editor, through an XML editing tool, or through a JSP page authoring tool that is aware of the XML syntax. Any JSP page author that is generating XML content should consider the use of JSP documents. In contrast, the XML view of a JSP page is a concept internal to the JSP container and is of interest only to Tag Library Authors and to implementors of JSP containers.

JSP.6.2 JSP Documents

A JSP document is a JSP page that is a namespace-aware XML document and that is identified as a JSP document to the JSP container.

JSP.6.2.1 Identifying JSP Documents

A JSP document can be identified as such in three ways:

- If there is a <jsp-property-group> that explicitly indicates, through the <is-xml> element, whether a given file is a JSP document, then that indication overrides any other determination. Otherwise,
- If this web application is using a version 2.4 web.xml, and if the extension is .jspx, then the file is a JSP document. Otherwise,
- If the file is explicitly or implicitly identified as a JSP page and the top element is a jsp:root element then the file is identified as a JSP document. This behavior provides backwards compatibility with JSP 1.2.

It is a translation-time error for a file that is identified as a JSP document to not be a well-formed, namespace-aware, XML document.

See Section JSP.8.6 for details on identifying tag files in XML syntax.

JSP.6.2.2 Overview of Syntax of JSP Documents

A JSP document may or not have a <jsp:root> as its top element; <jsp:root> was mandatory in JSP 1.2, but we expect most JSP documents in JSP 2.0 not to use it.

JSP documents identify standard actions through the use of a well-defined URI in its namespace; although in this chapter the prefix jsp is used for the standard actions, any prefix is valid as long as the correct URI identifying JSP 2.0 standard actions is used. Custom actions are identified using the URI that identifies their tag library; taglib directives are not required and cannot appear in a JSP document.

A JSP document can use XML elements as template data; these elements may have qualified names (and thus be in a namespace), or be unqualified.

The <jsp:text> element can be used to define some template data verbatim.

Since a JSP document must be a valid XML document, there are some JSP elements that can't be used in a JSP document. The elements that can be used are:

- JSP directives and scripting elements in XML syntax.
- EL expressions in the body of elements and in attribute values.
- All JSP standard actions described in Chapter JSP.1.
- The jsp:root, jsp:text, and jsp:output elements.
- Custom action elements
- Template data described using jsp:text elements.
- Template data described through XML fragments.

Scriptlet expressions used to specify request-time attribute values use a slightly different syntax in JSP documents than in non JSP documents; rather than using <%= expr %>, they use %= expr %. The white space around expr is not needed, and note the missing < and >. The expr, after any applicable quoting as in any other XML document, is an expression to be evaluated as in Section JSP.1.14.1.

The mechanisms that enable scripting and EL evaluation in a JSP page apply also when the page is a JSP document. Just as in the standard syntax, the $ in an EL expression can be quoted using \$ in both attribute values and template text. Recall, however, that \\ is not an escape sequence in XML attributes so whereas within an attribute in standard syntax \\${1+1} would result in \2 (assuming EL is enabled) or \${1+1} (assuming EL is ignored), in XML syntax \\${1+1} always results in \${1+1}.

It should be noted that the equivalent JSP document form of <a href="<%= url %>">, where 'a' is not a custom action, is:

```
<jsp:text><![CDATA[<a href="]]></jsp:text><jsp:expression>url</jsp:expres-
sion><jsp:text><![CDATA[">]]></jsp:text>
```

In the JSP document element <a href="%= url %">, "%= url %" does not represent a request-time attribute value. That syntax only applies for custom action elements. This is in contrast to <a href="${url}">, where "${url}" represents an EL expression in both JSP pages and JSP documents.

JSP.6.2.3 Semantic Model

The semantic model of a JSP document is unchanged from that of a JSP page in JSP syntax: JSP pages generate a response stream of characters from template data and dynamic elements. Template data can be described explicitly through a jsp:text element, or implicitly through an XML fragment. Dynamic elements are EL expressions, scripting elements, standard actions or custom actions. Scripting elements are represented as XML elements with the exception of request-time attribute expressions, which are represented through special attribute syntax.

The first step in processing a JSP document is to process it as an XML document, checking for well-formedness, processing entity resolution and, if applicable, performing validation as described in Section JSP.6.2.4. As part of the processing XML quoting will be performed, and JSP quoting will not be performed later.

After these steps, the JSP document will be passed to the JSP container which will then interpret it as a JSP page.

The JSP processing step for a JSP document is as for any other JSP page except that namespaces are used to identify standard actions and custom action tag libraries and that run time expressions in attributes use the slightly different syntax. Note that all the JSP elements that are described in this chapter are valid in all JSP pages, be they identified as JSP documents or not. This is a backward compatible change from the behavior in JSP 1.2 to enable gradual introduction of XML syntax in existing JSP pages.

To clearly explain the processing of whitespace, we follow the structure of the XSLT specification. The first step in processing a JSP document is to identify the nodes of the document. Then, all textual nodes that have only white space are dropped from the document; the only exception are nodes in a jsp:text element, which are kept verbatim. The resulting nodes are interpreted as described in the following sections. Template data is either passed directly to the response or it is mediated through (standard or custom) actions.

Following the XML specification (and the XSLT specification), whitespace characters are #x20, #x9, #xD, or #xA.

The container will add, in some conditions, an XML declaration to the output; the rules for this depend on the use of jsp:root and jsp:output; see Section JSP.6.3.3.

JSP.6.2.4 JSP Document Validation

A JSP Document with a DOCTYPE declaration must be validated by the container in the translation phase. Validation errors must be handled the same way as any other translation phase errors, as described in Section JSP.1.4.1.

JSP 2.0 requires only DTD validation for JSP Documents; containers should not perform validation based on other types of schemas, such as XML schema.

JSP.6.3 Syntactic Elements in JSP Documents

This section describes the elements in a JSP document.

JSP.6.3.1 Namespaces, Standard Actions, and Tag Libraries

JSP documents and tag files in XML syntax use XML namespaces to identify the standard actions, the directives, and the custom actions. JSP pages and tags in the JSP syntax cannot use XML namespaces and instead must use the taglib directive.

Though the prefix "jsp" is used throughout this specification, it is the namespace http://java.sun.com/JSP/Page and not the prefix "jsp" that identifies the JSP standard actions.

An xmlns attribute for a custom tag library of the form *xml:prefix='uri'* identifies the tag library through the uri value. The uri value may be of one of three forms, either a URN of the form urn:jsptagdir:*tagdir*, a URN of the form urn:jsptld:*path*, or a plain URI.

If the uri value is a URN of the form urn:jsptld:*path*, then the TLD is determined following the mechanism described in Section JSP.7.3.2.

If the uri value is a URN of the form urn:jsptagdir:*tagdir*, then the TLD is determined following the mechanism described in Section JSP.8.4.

If the uri value is a plain URI, then a path is determined by consulting the mapping indicated in web.xml extended using the implicit maps in the packaged tag libraries (Sections JSP.7.3.3 and JSP.7.3.4), as indicated in Section JSP.7.3.6. In contrast to Section JSP.7.3.6.2, however, a translation error must not be generated if the given uri is not found in the taglib map. Instead, any actions in the namespace defined by the uri value must be treated as uninterpreted.

JSP.6.3.2 The jsp:root Element

The jsp:root element can only appear as the root element in a JSP document or in a tag file in XMLsyntax; otherwise a translation error shall occur. JSP documents and tag files in XML syntax need not have a jsp:root element as its root element.

The jsp:root element has two main uses. One is to indicate that the JSP file is in XML syntax, without having to use configuration group elements nor using the .jspx extension. The other use of the jsp:root element is to accomodate the generation of content that is not a single XML document: either a sequence of XML documents or some non-XML content.

A jsp:root element can be used to provide zero or more xmlns attributes that correspond to namespaces for the standard actions, for custom actions or for generated template text. Unlike in JSP 1.2, not all tag libraries used within the JSP document need to be introduced on the root; tag libraries can be incorporated as needed inside the document using additional xmlns attributes.

The jsp:root element has one mandatory element, the version of the JSP spec that the page is using.

When jsp:root is used, the container will, by default, not insert an XML declaration; the default can be changed using the jsp:output element.

Examples

The following example generates a sequence of two XML documents. No XML declaration is generated.

```
<jsp:root xmlns:jsp=”http://java.sun.com/JSP/Page” version=”2.0">
    <table>foo</table>
    <table>bar</table>
</jsp:root>
```

The following example generates one XML document. An XML declaration is generated because of the use of jsp:output. The example is mostly instructional, as the same content could be generated dropping the jsp:root element.

```
<jsp:root xmlns:jsp=”http://java.sun.com/JSP/Page” version=”2.0">
    <jsp:output omit-xml-declaration="no"/>
    <table>foo</table>
</jsp:root>
```

Syntax

The root element has one mandatory attribute, the version of the JSP specification the page is using. No other attributes are defined in this element.

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
        body...
</jsp:root>
```

The one valid, mandatory, attribute of jsp:root is the version of the JSP specification used:

Table JSP.6-2 *Attributes for the <jsp:root> standard action*

version	(required) The version of the JSP specification used in this page. Valid values are "1.2" and "2.0". It is a translation error if the container does not support the specified version.

JSP.6.3.3 The jsp:output Element

The jsp:output element can be used in JSP documents and in tag files in XML syntax. The jsp:output element is described in detail in Section JSP.5.16.

JSP.6.3.4 The jsp:directive.page Element

The jsp:directive.page element defines a number of page dependent properties and communicates these to the JSP container. This element must be a child of the root element. Its syntax is:

<jsp:directive.page *page_directive_attr_list* />

Where page_directive_attr_list is as described in Section JSP.1.10.1.

The interpretation of a jsp:directive.page element is as described in Section JSP.1.10.1, and its scope is the JSP document and any fragments included through an include directive.

JSP.6.3.5 The jsp:directive.include Element

The jsp:directive.include element is used to substitute text and/or code at JSP page translation-time. This element can appear anywhere within a JSP document. Its syntax is:

<jsp:directive.include file="*relativeURLspec*" />

The interpretation of a jsp:directive.include element is as in Section JSP.1.10.3.

The XML view of a JSP page does not contain jsp:directive.include elements, rather the included file is expanded in-place. This is done to simplify validation.

JSP.6.3.6 Additional Directive Elements in Tag Files

Chapter JSP.8 describes the tag, attribute and variable directives, which can be used in tag files. The XML syntax for these directives is the same as in the XML view (see Section JSP.10.1.14, Section JSP.10.1.15, and Section JSP.10.1.16 for details).

JSP.6.3.7 Scripting Elements

The usual scripting elements: declarations, scriptlets and expressions, can be used in JSP documents, but the only valid forms for these elements in a JSP document are the XML syntaxes; i.e. those using the elements jsp:declaration, jsp:scriptlet and jsp:expression.

The jsp:declaration element is used to declare scripting language constructs that are available to all other scripting elements. A jsp:declaration element has no

attributes and its body is the declaration itself. The interpretation of a jsp:declaration element is as in Section JSP.1.12.1. Its syntax is:

```
<jsp:declaration> declaration goes here </jsp:declaration>
```

The jsp:scriptlet element is used to describe actions to be performed in response to some request. Scriptlets are program fragments. A jsp:scriptlet element has no attributes and its body is the program fragment that comprises the scriptlet. The interpretation of a jsp:scriptlet element is as in Section JSP.1.12.2. Its syntax is:

```
<jsp:scriptlet> code fragment goes here </jsp:scriptlet>
```

The jsp:expression element is used to describe complete expressions in the scripting language that get evaluated at response time. A jsp:expression element has no attributes and its body is the expression. The interpretation of a jsp:expression element is as in Section JSP.1.12.3. Its syntax is:

```
<jsp:expression> expression goes here </jsp:expression>
```

JSP.6.3.8 Other Standard Actions

The standard actions of Chapter 5 use a syntax that is consistent with XML syntax and they can be used in JSP documents and in tag files in XML syntax.

JSP.6.3.9 Template Content

A JSP page has no structure on its template content, and, correspondingly, imposes no constraints on that content. On the other hand, JSP documents have structure and some constraints are needed.

JSP documents can generate unconstrained content using jsp:text, as defined in Section JSP.5.15. Jsp:text can be used to generate totally fixed content but it can also be used to generate some dynamic content, as described in Section JSP.6.3.10 below.

Fixed structured content can be generated using XML fragments. A template XML element, an element that represents neither a standard action nor a custom action, can appear anywhere where a jsp:text may appear in a JSP document. The interpretation of such an XML element is to pass its textual representation to the current value of out, after the whitespace processing described in Section JSP.6.2.3.

For example, if the variable *i* has the value *3*, and the JSP document is of the form. :

Table JSP.6-3 *Example 1 - Input*

LineNo	Source Text
1	`<hello>`
2	`<hi>`
3	`<jsp:text> hi you all`
4	`</jsp:text>${i}`
5	`</hi>`
6	`</hello>`

The result is:

Table JSP.6-4 *Example 1 - Output*

LineNo	Output Text
1	`<hello><hi> hi you all`
2	`3</hi></hello>`

JSP.6.3.10 Dynamic Template Content

Custom actions can be used to generate any content, both structured and unstructured. Future versions of the JSP specification may allow for custom actions to check constraints on the generated content (see Section JSP.6.5.1) but the current specification has no standards support for any such constraints.

The most flexible standard mechanism for dynamic content is jsp:element. jsp:element, together with jsp:attribute and jsp:body can be used to generate any element. Further details of jsp:element, jsp:attribute and jsp:body are given in Section JSP.5.14, in Section JSP.5.10 and in Section JSP.5.11. The following example is from that section

```
<jsp:element
        name="${content.headerName}"
        xmlns:jsp="http://java.sun.com/JSP/Page">
    <jsp:attribute name="lang">${content.lang}</jsp:attribute>
    <jsp:body>${content.body}</jsp:body>
</jsp:element>
```

In some cases, the dynamic content that is generated can be described as simple substitutions on otherwise static templates. JSP documents can have XML

templates where EL expressions are used as the values of the body or of attributes. For instance, the next example uses the expression *table.indent* as the value of an attribute, and the expression *table.value* as that for the body of an element:

```
<table indent="${table.indent}">
    <row>${table.value}</row>
</table>
```

JSP.6.4 Examples of JSP Documents

The following sections provide several annotated examples of JSP documents.

JSP.6.4.1 Example: A simple JSP document

This simple JSP document generates a table with 3 rows with numeric values 1, 2, 3. The JSP document uses template XML elements intermixed with actions from the JSP Standard Tag Library.

```
<table size="${3}">
    <c:forEach
        xmlns:c="http://java.sun.com/jsp/jstl/core"
        var="counter" begin="1" end="${3}">
            <row>${counter}</row>
    </c:forEach>
</table>
```

Some comments:

- The XML template elements are *<table>* and *<row>*. The custom action element is *<c:forEach>*
- The JSP standard tag library is introduced through the use of its URI namespace and the specific prefix used, *c* in this case, is irrelevant. The prefix is introduced in a non-root element, and the top element of the document is still *<table>*.
- The expression *${counter}* is used within the *<row>* template element.
- The expression *${3}* (3 would have been equally good, but an expression is used for expository reasons) is used within the value of an attribute in both the XML template element *<table>* and in the custom action element *<c:forEach>*.

- The JSP document does not have an xml declaration - we are assuming the encoding of the file did not require it, e.g. it used UTF-8, - but the output will include an xml declaration due to the defaulting rules and to the absence of *jsp:output* element directing the container to do otherwise.

  The JSP document above does not generate an XML document that uses namespaces, but the next example does.

JSP.6.4.2 Example: Generating Namespace-aware documents

```
<table
    xmlns="http://table.com/Table1"
    size="${3}">
    <c:forEach
        xmlns:c="http://java.sun.com/jsp/jstl/core"
        var="counter" begin="1" end="${3}">
            <row>${counter}</row>
    </c:forEach>
</table>
```

This example is essentially the same as the one above, except that a default namespace is introduced in the top element The namespace applies to the unqualified elements: <*table*> and <*row*>. Also note that if the default namespace were to correspond to a custom action, then the elements so effected would be interpreted as invocations on custom actions or tags.

Although the JSP container understands that this document is a namespace-aware document. the JSP 2.0 container does not really understand that the generated content is a well-formed XML document and, as the next example shows, a JSP document can generate other types of content.

JSP.6.4.3 Example: Generating non-XML documents

```
<jsp:root
    xmlns:c="http://java.sun.com/jsp/jstl/core"
    xmlns:jsp="http://java.sun.com/JSP/Page"
    version="2.0">
    <c:forEach
        var="counter" begin="1" end="${3}">
            <jsp:text>${counter}</jsp:text>
    </c:forEach>
</jsp:root>
```

This example just generates *123*. There is no xml declaration generated because there is no <*jsp:output*> element to modify the default rule for when a JSP document has <*jsp:root*>. No additional whitespace is introduced because there is none within the <*jsp:text*> element.

The previous example used elements in the JSP namespace. That example used the *jsp* prefix, but, unlike with JSP pages in JSP syntax, the name of the prefix is irrelevant (although highly convenient) in JSP documents: the JSP URI is the only important indicative and the corrent URI should be used, and introduced via a namespace attribute.

For example, the same output would be generated with the following modification of the previous example:

```
<wombat:root
    xmlns:c="http://java.sun.com/jsp/jstl/core"
    xmlns:wombat="http://java.sun.com/JSP/Page"
    version="2.0">
    <c:forEach
        var="counter" begin="1" end="${3}">
            <wombat:text>${counter}</wombat:text>
    </c:forEach>
</wombat:root>
```

On the other hand, although the following example uses the *jsp* prefix the URI used in the namespace attribute is not the JSP URI and the JSP document will generate as output an XML document with root <*jsp:root*> using the URI *"http://johnsonshippingproducts.com"*.

```
<jsp:root
    xmlns:c="http://java.sun.com/jsp/jstl/core"
    xmlns:jsp="http://johnsonshippingproducts.com"
    version="2.0">
    <c:forEach
        var="counter" begin="1" end="${3}">
            <jsp:text>${counter}</jsp:text>
    </c:forEach>
</jsp:root>
```

Finally, note that, since a JSP document is a well-formed, namespace-aware document, prefixes, including *jsp* cannot be used without being introduced through a namespace attribute.

JSP.6.4.4 Example: Using Custom Actions and Tag Files

Custom actions are frequently used within a JSP document to generate portions of XML content. The JSP specification treats this content as plain text, with no intepretation nor constraints imposed on it. Good practice, though, suggests abstractions that organize the content along well-formed fragments.

The following example generates an XHTML document using tag library abstractions for presentation and data access, made available through the prefixes *u* and *data* respectively.

```
<html    xmlns="http://www.w3.org/1999/xhtml"
         xmlns:jsp="http://java.sun.com/JSP/Page"
         xmlns:c="http://java.sun.com/jsp/jstl/core"
         xmlns:u="urn:jsptagdir:/WEB-INF/tags/mylib/"
         xmlns:data="http://acme.com/functions">
    <c:set var="title" value="Best Movies" />
    <u:headInfo title="${title}"/>
    <body>
        <h1>${title}</h1>
        <h2>List of Best Movies</h2>
        <ul>
            <c:forEach var="m" varStatus="s" items="data:movieItems()">
                <li><a href="#EL${s.index}">${s.index}</a>${m.title}</li>
            </c:forEach>
        </ul>
    </body>
</html>
```

For convenience we use the *<c:set>* JSTL action, which defines variables and associates values with them. This allows grouping in a single place of definitions used elsewhere.

The action *<u:headInfo>* could be implemented either through a custom action or through a tag. For example, as a tag it could be defined by the following code:

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
    <jsp:directive.tag />
    <jsp:directive.attribute name=”title” required="true" />
    <head>
        <meta   http-equiv="content-type"
                content="text/html;charset=${pageCharSet}" />
        <title>${title}</title>
    </head>
</jsp:root>
```

where *pageCharSet* is a variable with a value as *iso-8859-1*.

Note that this tag is a JSP document (because of the jsp:root declaration), and, as such, it is validated by the container. Also note that the content that is generated in this case is not using QNames, which means that the interpretation of the generated elements can be ’captured’ based on the invocation context. That is the case here, as there is a default namespace active (that of XHTML) where the tag is being invoked.

JSP.6.5 Possible Future Directions for JSP documents

This section is non-normative. Two features are sketched briefly here to elicit input that could be used on future versions of the JSP specification.

JSP.6.5.1 Generating XML Content Natively

All JSP 2.0 content is textual, even when using JSP documents to generate XML content. This is quite acceptable, and even ideal, for some applications, but in some other applications XML documents are the main data type being manipulated. For example, the data source may be an XML document repository, perhaps queried using XQuery, some of the manipulation on this data internal to the JSP page will use XML concepts (XPath, XSTL operations), and the generated XML document may be part of some XML pipeline.

In one such application, it is appealing not to transform back and forth between a stream of characters (text) and a parsed representation of the XML document. The JSP expert group has explored different approaches on how such XML-awareness could be added, and a future version of JSP could support this functionality.

JSP.6.5.2 Schema and XInclude Support

The current specification only requires DTD validation support for JSP documents. A more flexible schema language, like XML Schema, could be useful and could be explored by a future version of the JSP specification.

Similarly, future versions of the specification may also consider support for XInclude.

CHAPTER JSP.7

Tag Extensions

This chapter describes the tag library facility for introducing new actions into a JSP page. The tag library facility includes portable run-time support, a validation mechanism, and authoring tool support. Both the classic JSP 1.2 style tag extension mechanism and the newer JSP 2.0 simple tag extension mechanism are described. In Chapter JSP.8, “Tag Files”, a mechanism for authoring tag extensions using only JSP syntax is described. This brings the power of tag extensions to page authors that may not know the Java programming language.

This chapter also provides an overview of the tag library concept. It describes the Tag Library Descriptor, and the taglib directive. A detailed description of the APIs involved follows in Chapter JSP.13.

JSP.7.1 Introduction

A Tag Library abstracts functionality used by a JSP page by defining a specialized (sub)language that enables a more natural use of that functionality within JSP pages.

The actions introduced by the Tag Library can be used by the JSP page author in JSP pages explicitly, when authoring the page manually, or implicitly, when using an authoring tool. Tag Libraries are particularly useful to authoring tools because they make intent explicit and the parameters expressed in the action instance provide information to the tool.

Actions that are delivered as tag libraries are imported into a JSP page using the taglib directive. They are available for use in the page using the prefix given by the directive. An action can create new objects that can be passed to other actions, or can be manipulated programmatically through a scripting element in the JSP page.

The semantics of a specific custom action in a tag library is described via a tag handler class which is usually instantiated at runtime by the JSP page implementation class. When the tag library is well known to the JSP container (Section JSP.7.3.9), the Container can use alternative implementations as long as the semantics are preserved.

Tag libraries are portable: they can be used in any legal JSP page regardless of the scripting language used in that page.

The tag extension mechanism includes information to:

- Execute a JSP page that uses the tag library.
- Author or modify a JSP page.
- Validate the JSP page.
- Present the JSP page to the end user.

A Tag Library is described via the Tag Library Descriptor (TLD), an XML document that is described below.

JSP.7.1.1 Goals

The tag extension mechanism described in this chapter addresses the following goals. It is designed to be:

- *Portable* - An action described in a tag library must be usable in any JSP container.
- *Simple* - Unsophisticated users must be able to understand and use this mechanism. Vendors of JSP functionality must find it easy to make it available to users as actions.
- *Expressive* - The mechanism must support a wide range of actions, including nested actions, scripting elements inside action bodies, and creation, use, and updating of scripting variables.
- *Usable from different scripting languages* - Although the JSP specification currently only defines the semantics for scripts in the Java programming language, we want to leave open the possibility of other scripting languages.
- *Built upon existing concepts and machinery* - We do not want to reinvent what exists elsewhere. Also, we want to avoid future conflicts whenever we can predict them.

JSP.7.1.2 Overview

The processing of a JSP page conceptually follows these steps:

Parsing

JSP pages can be authored using two different syntaxes: a JSP syntax and an XML syntax. The semantics and validation of a JSP syntax page is described with reference to the semantics and validation of an equivalent document in the XML syntax.

The first step is to parse the JSP page. The page that is parsed is as expanded by the processing of include directives. Information in the TLD is used in this step, including the identification of custom tags, so there is some processing of the taglib directives in the JSP page.

Validation

The tag libraries in the XML document are processed in the order in which they appear in the page.

Each library is checked for a validator class. If one is present, the whole document is made available to its validate method as a PageData object. As of JSP 2.0, the Container must provide a jsp:id attribute. This information can be used to provide location information on errors.

Each custom tag in the library is checked for a TagExtraInfo class. If one is present, its validate method is invoked. The default implementation of validate is to call isValid. See the APIs for more details.

Translation

Finally, the XML document is processed to create a JSP page implementation class. This process may involve creating scripting variables. Each custom action will provide information about variables, either statically in the TLD, or more flexibly by using the getVariableInfo method of a TagExtraInfo class.

Execution

Once a JSP page implementation class has been associated with a JSP page, the class will be treated as any other servlet class: Requests will be directed to instances of the class. At run-time, tag handler instances will be created and methods will be invoked in them.

JSP.7.1.3 Classic Tag Handlers

A classic tag handler is a Java class that implements the Tag, IterationTag, or BodyTag interface, and is the run-time representation of a custom action.

The JSP page implementation class instantiates a tag handler object, or reuses an existing tag handler object, for each action in the JSP page. The handler object is a Java object that implements the javax.servlet.jsp.tagext.Tag interface. The handler object is responsible for the interaction between the JSP page and additional server-side objects.

There are three main interfaces: Tag, IterationTag, and BodyTag.

- The Tag interface defines the basic methods needed in all tag handlers. These methods include setter methods to initialize a tag handler with context data and attribute values of the action, and the doStartTag and doEndTag methods.
- The IterationTag interface is an extension to Tag that provides the additional method, doAfterBody, invoked for the reevaluation of the body of the tag.
- The BodyTag interface is an extension of IterationTag with two new methods for when the tag handler wants to manipulate the tag body: setBodyContent passes a buffer, the BodyContent object, and doInitBody provides an opportunity to process the buffer before the first evaluation of the body into the buffer.

The use of interfaces simplifies making an existing Java object a tag handler. There are also two support classes that can be used as base classes: TagSupport and BodyTagSupport.

JSP 1.2 introduced a new interface designed to help maintain data integrity and resource management in the presence of exceptions. The TryCatchFinally interface is a “mix-in” interface that can be added to a class implementing any of Tag, IterationTag, or BodyTag.

JSP.7.1.4 Simple Examples of Classic Tag Handlers

As examples, we describe prototypical uses of tag extensions, briefly sketching how they take advantage of these mechanisms.

JSP.7.1.4.1 Plain Actions

The simplest type of action just *does something*, perhaps with parameters to modify what the “something” is, and improve reusability.

This type of action can be implemented with a tag handler that implements the Tag interface. The tag handler needs to use only the doStartTag method which is

invoked when the start tag is encountered. It can access the attributes of the tag and information about the state of the JSP page. The information is passed to the Tag object through setter method calls, prior to the call to doStartTag.

Since simple actions with empty tag bodies are common, the Tag Library Descriptor can be used to indicate that the tag is always intended to be empty. This indication leads to better error checking at translation time, and to better code quality in the JSP page implementation class.

JSP.7.1.4.2 Actions with a Body

Another set of simple actions require something to happen when the start tag is found, and when the end tag is found. The Tag interface can also be used for these actions. The doEndTag is similar to the doStartTag method except that it is invoked when the end tag of the action is encountered. The result of the doEndTag invocation indicates whether the remainder of the page is to be evaluated or not.

JSP.7.1.4.3 Conditionals

In some cases, a body needs to be invoked only when some (possibly complex) condition happens. Again, this type of action is supported by the basic Tag interface through the use of return values in the doStartTag method.

JSP.7.1.4.4 Iterations

For iteration the IterationTag interface is needed. The doAfterBody method is invoked to determine whether to reevaluate the body or not.

JSP.7.1.4.5 Actions that Process their Body

Consider an action that evaluates its body many times, creating a stream of response data. The IterationTag protocol is used for this.

If the result of the reinterpretation is to be further manipulated for whatever reason, including just discarding it, we need a way to divert the output of reevaluations. This is done through the creation of a BodyContent object and use of the setBodyContent method, which is part of the BodyTag interface. BodyTag also provides the doInitBody method which is invoked after setBodyContent and before the first body evaluation provides an opportunity to interact with the body.

JSP.7.1.4.6 Cooperating Actions

Cooperating actions may offer the best way to describe a desired functionality. For example, one action may be used to describe information leading to the creation

of a server-side object, while another action may use that object elsewhere in the page. These actions may cooperate explicitly, via scoped variables: one action creates an object and gives it a name; the other refers to the object through the name.

Two actions can also cooperate implicitly. A flexible and convenient mechanism for action cooperation uses the nested structure of the actions to describe scoping. This is supported in the specification by providing each tag handler with its parent tag handler (if any) through the setParent method. The findAncestorWithClass static method in TagSupport can then be used to locate a tag handler, and, once located, to perform valid operations on the tag handler.

JSP.7.1.4.7 Actions Defining Scripting Variables

A custom action may create server-side objects and make them available to scripting elements by creating or updating the scripting variables. The variables thus affected are part of the semantics of the custom action and are the responsibility of the tag library author.

This information is used at JSP page translation time and can be described in one of two ways: directly in the TLD for simple cases, or through subclasses of TagExtraInfo. Either mechanism will indicate the names and types of the scripting variables.

At request time the tag handler will associate objects with the scripting variables through the pageContext object.

It is the responsibility of the JSP page translator to automatically supply the code required to do the "synchronization" between the pageContext values and the scripting variables.

There are some sections of JSP where scripting is not allowed. For example, this is the case in a tag body where the body-content is declared as 'scriptless', or in a page where <scripting-invalid> is true. In these sections, it is not possible to access scripting variables directly via scriptlets or expressions, and therefore the container need not synchronize them. Instead, the page author can use the EL to access the pageContext values.

JSP.7.1.5 Simple Tag Handlers

The API and invocation protocol for classic tag handlers is necessarily somewhat complex because scriptlets and scriptlet expressions in tag bodies can rely on surrounding context defined using scriptlets in the enclosing page.

With the advent of the Expression Language (EL) and JSP Standard Tag Library (JSTL), it is now feasible to develop JSP pages that do not need scriptlets

or scriptlet expressions. This allows us to define a tag invocation protocol that is easier to use for many use cases.

In that interest, JSP 2.0 introduces a new type of tag extension called a Simple Tag Extension. Simple Tag Extensions can be written in one of two ways:

- In Java, by defining a class that implements the javax.servlet.jsp.tagext.SimpleTag interface. This class is intended for use by advanced page authors and tag library developers who need the flexibility of the Java language in order to write their tag handlers. The javax.servlet.jsp.tagext.SimpleTagSupport class provides a default implementation for all methods in SimpleTag.
- In JSP, using tag files. This method can be used by page authors who do not know Java. It can also be used by advanced page authors or tag library developers who know Java but are producing tag libraries that are presentation-centric or can take advantage of existing tag libraries. See Chapter JSP.8, “Tag Files” for more details.

The lifecycle of a Simple Tag Handler is straightforward and is not complicated by caching semantics. Once a Simple Tag Handler is instantiated by the Container, it is executed and then discarded. The same instance must not be cached and reused. Initial performance metrics show that caching a tag handler instance does not necessarily lead to greater performance, and to accommodate such caching makes writing portable tag handlers difficult and makes the tag handler prone to error.

In addition to being simpler to work with, Simple Tag Extensions do not directly rely on any servlet APIs, which allows for potential future integration with other technologies. This is facilitated by the JspContext class, which PageContext now extends. JspContext provides generic services such as storing the JspWriter and keeping track of scoped attributes, whereas PageContext has functionality specific to serving JSPs in the context of servlets. Whereas the Tag interface relies on PageContext, SimpleTag only relies on JspContext.

The body of a Simple Tag, if present, is translated into a JSP Fragment and passed to the setJspBody method. The tag can then execute the fragment as many times as needed. See Section JSP.7.1.6 for more details on JSP Fragments. Because JSP fragments do not support scriptlets, the <body-content> of a SimpleTag cannot be “JSP”. A TLD is invalid if it specifies “JSP” as the value for <body-content> for a tag whose handler implements the SimpleTag interface. JSP containers are recommended to but not required to produce an error if “JSP” is specified in this case.

JSP.7.1.6 JSP Fragments

During the translation phase, various pieces of the page are translated into implementations of the javax.servlet.jsp.tagext.JspFragment abstract class, before being passed to a tag handler. This is done automatically for any JSP code in the body of a *named attribute* (one that is defined by <jsp:attribute>) that is declared to be a fragment, or of type JspFragment, in the TLD. This is also automatically done for the body of any tag handled by a Simple Tag handler. Once passed in, the tag handler can then evaluate and re-evaluate the fragment as many times as needed, or even pass it along to other tag handlers, in the case of Tag Files.

A JSP fragment can be parameterized by a tag handler by setting page-scoped attributes in the JspContext associated with the fragment. These attributes can then be accessed via the EL.

A translation error must occur if a piece of JSP code that is to be translated into a JSP Fragment contains scriptlets or scriptlet expressions.

See Chapter JSP.13, “Tag Extension API” for more details on the JspFragment abstract class.

JSP.7.1.7 Simple Examples of Simple Tag Handlers

In this section, we revisit the prototypical uses of classic tag extensions, as was presented in Section JSP.7.1.4, and briefly describe how they are implemented using simple tag handlers.

JSP.7.1.7.1 Plain Actions

To implement plain actions, the tag library developer creates a class that extends SimpleTagSupport and implements the doTag method. The details on accessing attributes and enforcing an empty body are the same as with classic tag handlers. By default, the rest of the page will be evaluated after invoking doTag. To signal that the page is to be skipped, doTag throws SkipPageException.

JSP.7.1.7.2 Actions with a Body

To implement actions with a body, the tag library developer implements doTag and invokes the body at any point by calling invoke on the JspFragment object passed in via the setJspBody method. The tag handler can provide the fragment access to variables through the JspContext object.

JSP.7.1.7.3 Conditionals

All conditional logic is handled in the doTag method. If the body is not to be invoked, the tag library developer simply does not call invoke on the JspFragment object passed in via setJspBody.

JSP.7.1.7.4 Iterations

All iteration logic is handled in the doTag method. The tag library developer simply calls invoke on the JspFragment object passed in via setJspBody as many times as needed.

JSP.7.1.7.5 Actions that Process their Body

To divert the result of the body invocation, the tag library developer passes a java.io.Writer object to the invoke method on the body JspFragment. Unlike the standard tag handler's BodyContent solution, the result of the invocation does not need to be buffered.

JSP.7.1.7.6 Cooperating Actions

Cooperating actions work the same way as with classic tag handlers. A setParent method is available in the SimpleTag interface and is called by the container before calling doTag if one tag invocation is nested within another. A findAncestorWithClass method is available on SimpleTagSupport. This should be used, instead of TagSupport.findAncestorWithClass(), in all cases where the desired return value may implement SimpleTag.

Note that SimpleTag does not extend Tag. Because of this, the JspTag common base is used in these new APIs instead of Tag. Furthermore, because Tag.setParent only accepts an object of type Tag, tag collaboration becomes more difficult when classic tag handlers are nested inside SimpleTag custom actions.

To make things easier, the javax.servlet.jsp.tagext.TagAdapter class can wrap any SimpleTag and expose it as if it were a Tag instace. The original JspTag can be retrieved through its getAdaptee method. Whenever calling the setParent method on a classic Tag in a case where the outer tag does not implement Tag, the JSP Container must construct a new TagAdapter and call setParent on the classic Tag passing in the adapter.

See Chapter JSP.13, "Tag Extension API" for more details on these APIs.

JSP.7.1.8 Attributes With Dynamic Names

Prior to JSP 2.0, the name of every attribute that a tag handler accepted was predetermined at the time the tag handler was developed. It is sometimes useful, however, to be able to define a tag handler that accepts attributes with dynamic names that are not known until the page author uses the tag. For example, it is time consuming and error-prone to anticipate what attributes a user may wish to pass to a tag that mimics an HTML element.

New to JSP 2.0 is the ability to declare that a tag handler accepts additional attributes with dynamic names. This is done by having the tag handler implement the javax.servlet.jsp.tagext.DynamicAttributes interface. See Chapter JSP.13, "Tag Extension API" for more details on this interface.

JSP.7.1.9 Event Listeners

A tag library may include classes that are event listeners (see the Servlet 2.4 specification). The listeners classes are listed in the tag library descriptor and the JSP container automatically instantiates them and registers them. A Container is required to locate all TLD files (see Section JSP.7.3.1 for details on how they are identified), read their listener elements, and treat the event listeners as extensions of those listed in web.xml.

The order in which the listeners are registered is undefined, but they are registered before application start.

JSP.7.2 Tag Libraries

A *tag library* is a collection of actions that encapsulate some functionality to be used from within a JSP page. A tag library is made available to a JSP page through a taglib directive that identifies the tag library via a URI (Universal Resource Identifier).

The URI identifying a tag library may be any valid URI as long as it can be used to uniquely identify the semantics of the tag library.

The URI identifying the tag library is associated with a *Tag Library Description* (TLD) file and with *tag handler* classes as indicated in Section JSP.7.3 below.

JSP.7.2.1 Packaged Tag Libraries

JSP page authoring tools and JSP containers are required to accept a tag library that is packaged as a JAR file. When deployed in a JSP container, the standard JAR

conventions described in the Servlet 2.4 specification apply, including the conventions for dependencies on extensions.

Packaged tag libraries must have at least one tag library descriptor file. The JSP 1.1 specification allowed only a single TLD, in META-INF/taglib.tld, but as of JSP 1.2 multiple tag libraries are allowed. See Section JSP.7.3.1 for how TLDs are identified.

Both Classic and Simple Tag Handlers (implemented either in Java or as tag files) can be packaged together.

JSP.7.2.2 Location of Java Classes

A tag library contains classes for instantiation at translation time and classes for instantiation at request time. The former include classes such as TagLibraryValidator and TagExtraInfo. The latter include tag handler and event listener classes.

The usual conventions for Java classes apply: as part of a web application, they must reside either in a JAR file in the WEB-INF/lib directory, or in a directory in the WEB-INF/classes directory.

A JAR containing packaged tag libraries must be dropped into the WEB-INF/lib directory to make its classes available at request time (and also at translation time, see Section JSP.7.3.7). The mapping between the URI and the TLD is explained further below.

JSP.7.2.3 Tag Library directive

The taglib directive in a JSP page declares that the page uses a tag library, uniquely identifies the tag library using a URI, and associates a tag prefix with usage of the actions in the library.

A JSP container maps the URI used in the taglib directive into a Tag Library Descriptor in two steps: it resolves the URI into a TLD resource path, and then derives the TLD object from the TLD resource path.

If the JSP container cannot locate a TLD resource path for a given URI, a fatal translation error shall result. Similarly, it is a fatal translation error for a URI attribute value to resolve to two different TLD resource paths.

It is a fatal translation error for the taglib directive to appear after actions using the prefix introduced by it.

JSP.7.3 The Tag Library Descriptor

The Tag Library Descriptor (TLD) is an XML document that describes a tag library. The TLD for a tag library is used by a JSP container to interpret pages that include taglib directives referring to that tag library. The TLD is also used by JSP page authoring tools that will generate JSP pages that use a library, and by authors who do the same manually.

The TLD includes documentation on the library as a whole and on its individual tags, version information on the JSP container and on the tag library, and information on each of the actions defined in the tag library.

The TLD may name a TagLibraryValidator class that can validate that a JSP page conforms to a set of constraints expected by the tag library.

Each action in the library is described by giving its name, the class of its tag handler, information on any scripting variables created by the action, and information on attributes of the action. Scripting variable information can be given directly in the TLD or through a TagExtraInfo class. For each valid attribute there is an indication about whether it is mandatory, whether it can accept request-time expressions, and additional information.

A TLD file is useful for providing information on a tag library. It can be read by tools without instantiating objects or loader classes. Our approach conforms to the conventions used in other J2EE technologies.

As of JSP 2.0, the format for the Tag Library Descriptor is represented in XML Schema. This allows for a more extensible TLD that can be used as a true single-source document.

JSP.7.3.1 Identifying Tag Library Descriptors

Tag library descriptor files have names that use the extension .tld, and the extension indicates a tag library descriptor file. When deployed inside a JAR file, the tag library descriptor files must be in the META-INF directory, or a subdirectory of it. When deployed directly into a web application, the tag library descriptor files must always be in the WEB-INF directory, or some subdirectory of it. TLD files should not be placed in /WEB-INF/classes or /WEB-INF/lib.

The XML Schema for a TLD document is http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd. See Section JSP.C.1, “XML Schema for TLD, JSP 2.0”.

Note that tag files, which collectively form tag libraries, may or may not have an explicitly defined TLD. In the case that they do not, the container generates an implicit TLD that can be referenced using the tagdir attribute of the taglib

directive. More details about identifying this implicit Tag Library Descriptor can be found in Chapter JSP.8, “Tag Files”.

JSP.7.3.2 TLD resource path

A URI in a taglib directive is mapped into a context relative path (as discussed in Section JSP.1.2.1). The context relative path is a URL without a protocol and host components that starts with / and is called the *TLD resource path*.

The TLD resource path is interpreted relative to the root of the web application and should resolve to a TLD file directly, or to a JAR file that has a TLD file at location META-INF/taglib.tld. If the TLD resource path is not one of these two cases, a fatal translation error will occur.

The URI describing a tag library is mapped to a TLD resource path though a *taglib map*, and a fallback interpretation that is to be used if the map does not contain the URI. The taglib map is built from an explicit taglib map in web.xml (described in Section JSP.7.3.3) that is extended with implicit entries deduced from packaged tag libraries in the web application (described in Section JSP.7.3.4), and implicit entries known to the JSP container. The fallback interpretation is targetted to a casual use of the mechanism, as in the development cycle of the Web Application; in that case the URI is interpreted as a direct path to the TLD (see Section JSP.7.3.6.2).

The following order of precedence applies (from highest to lowest) when building the taglib map (see the following sections for details):

1. Taglib Map in web.xml
2. Implicit Map Entries from TLDs
   - TLDs in JAR files in WEB-INF/lib
   - TLDs under WEB-INF
3. Implicit Map Entries from the Container

JSP.7.3.3 Taglib Map in web.xml

The web.xml file can include an explicit taglib map between URIs and TLD resource paths described using the taglib elements of the Web Application Deployment descriptor in WEB-INF/web.xml. See Section JSP.3.2 for more details.

JSP.7.3.4 Implicit Map Entries from TLDs

The taglib map described in web.xml is extended with new entries extracted from TLD files in the Web Application. The new entries are computed as follows:

- The container searches for all files with a .tld extension under /WEB-INF or a subdirectory, and inside JAR files that are in /WEB-INF/lib. When examining a JAR file, only resources under /META-INF or a subdirectory are considered. The order in which these files are searched for is implementation-specific and should not be relied on by web applications.
- Each TLD file is examined. If it has a <uri> element, then a new <taglib> element is created, with a <taglib-uri> subelement whose value is that of the <uri> element, and with a <taglib-location> subelement that refers to the TLD file.
- If the created <taglib> element has a different <taglib-uri> to any in the taglib map, it is added.

This mechanism provides an automatic URI to TLD mapping as well as supporting multiple TLDs within a packaged JAR. Note that this functionality does not require explicitly naming the location of the TLD file, which would require a mechanism like the jar:protocol.

Note also that the mechanism does not add duplicated entries.

JSP.7.3.5 Implicit Map Entries from the Container

The Container may also add additional entries to the taglib map. As in the previous case, the entries are only added for URIs that are not present in the map. Conceptually the entries correspond to TLD describing these tag libraries.

These implicit map entries correspond to libraries that are known to the Container, who is responsible for providing their implementation, either through tag handlers, or via the mechanism described in Section JSP.7.3.9.

JSP.7.3.6 Determining the TLD Resource Path

The TLD resource path can be determined from the uri attribute of a taglib directive as described below. In the explanation below an absolute URI is one that starts with a protocol and host, while a relative URI specification is as in section 2.5.2, i.e. one without the protocol and host part.

All steps are described as if they were taken, but an implementation can use a different implementation strategy as long as the result is preserved.

JSP.7.3.6.1 Computing TLD Locations

The taglib map generated in Sections JSP.7.3.3, JSP.7.3.4 and JSP.7.3.5 may contain one or more <taglib></taglib> entries. Each entry is identified by a taglib_uri, which is the value of the <taglib-uri> subelement. This taglib_uri may be an absolute URI, or a relative URI spec starting with / or one not starting with /. Each entry also defines a taglib_location as follows:

- If the <taglib-location> subelement is some relative URI specification that starts with a / the taglib_location is this URI.
- If the <taglib-location> subelement is some relative URI specification that does not start with /, the taglib_location is the resolution of the URI relative to /WEB-INF/web.xml (the result of this resolution is a relative URI specification that starts with /).

JSP.7.3.6.2 Computing the TLD Resource Path

The following describes how to resolve a taglib directive to compute the TLD resource path. It is based on the value of the uri attribute of the taglib directive.

- If uri is abs_uri, an absolute URI

Look in the taglib map for an entry whose taglib_uri is abs_uri. If found, the corresponding taglib_location is the TLD resource path. If not found, a translation error is raised.

- If uri is root_rel_uri, a relative URI that starts with /

Look in the taglib map for an entry whose taglib_uri is root_rel_uri. If found, the corresponding taglib_location is the TLD resource path. If no such entry is found, root_rel_uri is the TLD resource path.

- If uri is noroot_rel_uri, a relative URI that does not start with /

Look in the taglib map for an entry whose taglib_uri is noroot_rel_uri. If found, the corresponding taglib_location is the TLD resource path. If no such entry is found, resolve noroot_rel_uri relative to the current JSP page where the directive appears; that value (by definition, this is a relative URI specification that starts with /) is the TLD resource path. For example, if /a/b/c.jsp references ../../WEB-INF/my.tld, then if there is no taglib_location that matches ../../WEB-INF/my.tld, the TLD resource path would be /WEB-INF/my.tld.

JSP.7.3.6.3 Usage Considerations

The explicit web.xml map provides a explicit description of the tag libraries that are being used in a web application.

The implicit map from TLDs means that a JAR file implementing a tag library can be dropped in and used immediatedly through its stable URIs.

The use of relative URI specifications in the taglib map enables very short names in the taglib directive. For example, if the map is:

```
<taglib>
  <taglib-uri>/myPRlibrary</taglib-uri>
  <taglib-location>/WEB-INF/tlds/PRlibrary_1_4.tld</taglib-location>
</taglib>
```

then it can be used as:

```
<%@ taglib uri="/myPRlibrary" prefix="x" %>
```

Finally, the fallback rule allows a taglib directive to refer directly to the TLD. This arrangement is very convenient for quick development at the expense of less flexibility and accountability. For example, in the case above, it enables:

```
<%@ taglib uri="/WEB-INF/tlds/PRlibrary_1_4.tld" prefix="x" %>
```

JSP.7.3.7 Translation-Time Class Loader

The set of classes available at translation time is the same as that available at runtime: the classes in the underlying Java platform, those in the JSP container, and those in the class files in WEB-INF/classes, in the JAR files in WEB-INF/lib, and, indirectly those indicated through the use of the class-path attribute in the META-INF/MANIFEST file of these JAR files.

JSP.7.3.8 Assembling a Web Application

As part of the process of assembling a web application, the Application Assembler will create a WEB-INF/ directory, with appropriate lib/ and classes/ subdirectories, place JSP pages, servlet classes, auxiliary classes, and tag libraries in the proper places, and create a WEB-INF/web.xml that ties everything together.

Tag libraries that have been delivered in the standard JAR format can be dropped directly into WEB-INF/lib. This automatically adds all the TLDs inside the JAR, making their URIs advertised in their <uri> elements visible to the URI to

TLD map. The assembler may create taglib entries in web.xml for each of the libraries that are to be used.

Part of the assembly (and later the deployment) may create and/or change information that customizes a tag library; see Section JSP.7.5.3.

JSP.7.3.9 Well-Known URIs

A JSP container may “know of” some specific URIs and may provide alternate implementations for the tag libraries described by these URIs, but the user must see the behavior as that described by the required, portable tag library description described by the URI.

A JSP container must always use the mapping specified for a URI in the web.xml deployment descriptor if present. If the deployer wants to use the platform-specific implementation of the well-known URI, the mapping for that URI should be removed at deployment time.

JSP.7.3.10 Tag and Tag Library Extension Elements

The JSP 2.0 Tag Library Descriptor supports the notion of Tag Extension Elements and Tag Library Extension Elements. These are elements added to the TLD by the tag library developer that provide additional information about the tag, using a schema defined outside of the JSP specification.

The information contained in these extensions is intended to be used by tools only, and is not accessible at compile-time, deployment-time, or run-time. JSP containers must not alter their behavior based on the content, the presence, or the absence of a particular Tag or Tag Library Extension Element. In addition, JSP containers must consider invalid any tag library that specifies mustUnderstand=”true” for any Tag or Tag Library Extension element. Any attempt to use an invalid tag library must produce a translation error. This is to preserve application compatibility across containers.

The JSP container may use schema to validate the structure of the Tag Library Descriptor. If it does so, any new content injected into Tag or Tag Library Extension elements must not be validated by the JSP Container.

Tag Library Extension Elements provide extension information at the tag library level, and are specified by adding a <taglib-extension> element as a child of <taglib>. Tag Extension Elements provide extension information at the tag level, and are specified by adding a <tag-extension> element as a child of <tag>. To use these elements, an XML namespace must first be defined and the namespace must be imported into the TLD.

There are efforts under way in the JCP (Java Community Process) to define standard extensions for enhanced tool support for JSP page authoring. Such standard extensions should be used where appropriate.

JSP.7.3.10.1 Example

In the following non-normative example, a fictitious company called ACME has decided to enhance the page author's experience by defining a set of Tag and Tag Library Extension elements that cause sounds to be played when inserting tags in a document.

In this hypothetical example, ACME has published an XML Schema at http://www.acme.com/acme.xsd that defines the extensions, and has provided plug-ins for various JSP-capable IDEs to recognize these extension elements.

The following example tag library uses ACME's extensions to provide helpful voice annotations that describe how to use each tag in the tag library. Relevant parts highlighted in bold:

```
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:acme="http://acme.com/"
       xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
           http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd
           http://acme.com/ http://acme.com/acme.xsd"
       version="2.0">
       <description>
           Simple Math Tag Library.
           Contains ACME sound extensions with helpful voice annotations
           that describe how to use the tags in this library.
       </description>
       <tlib-version>1.0</tlib-version>
       <short-name>math</short-name>
       <tag>
```

```
            <description>Adds two numbers</description>
            <display-name>add</display-name>
            <name>add</name>
            <tag-class>com.foobar.tags.AddTag</tag-class>
            <body-content>empty</body-content>
            <attribute>
                <name>x</name>
                <type>java.lang.Double</type>
            </attribute>
            <attribute>
                <name>y</name>
                <type>java.lang.Double</type>
            </attribute>
            <tag-extension namespace="http://acme.com/">
                <!-- Extensions for tag sounds -->
                <extension-element xsi:type="acme:acme-soundsType">
                    <acme:version>1.5</acme:version>
                    <!-- Sound played for help on the add tag -->
                    <acme:tag-sound>sounds/add.au</acme:tag-sound>
                    <!-- Sound played for help on the x attribute -->
                    <acme:attribute-sound name="x">
                        sounds/add-x.au
                    </acme:attribute-sound>
                    <!-- Sound that's played for help on the yattribute -->
                    <acme:attribute-sound name="y">
                        sounds/add-y.au
                    </acme:attribute-sound>
                </extension-element>
            </tag-extension>
        </tag>
        <taglib-extension namespace="http://acme.com/">
            <!-- Extensions for taglibrary sounds-->
            <extension-element xsi:type="acme:acme-soundsType">
                <acme:version>1.5</acme:version>
                <!-- Sound played when author imports this taglib -->
                <acme:import-sound>sounds/intro.au</acme:import-sound>
            </extension-element>
        </taglib-extension>
</taglib>
```

The corresponding acme.xsd file would look something like:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<xsd:schema
     targetNamespace="http://acme.com/"
     xmlns:j2ee="http://java.sun.com/xml/ns/j2ee"
     xmlns:acme="http://acme.com/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:xml="http://www.w3.org/XML/1998/namespace"
     elementFormDefault="qualified"
     attributeFormDefault="unqualified"
     version="1.0">

  <xsd:annotation>
    <xsd:documentation>
      This an XML Schema for sample Acme taglib extensibility
      elements, used to test TLD extensibility.
    </xsd:documentation>
  </xsd:annotation>

  <!-- ************************************************* -->

  <xsd:import namespace="http://java.sun.com/xml/ns/j2ee"
              schemaLocation="../web-jsptaglibrary_2_0.xsd" />

  <!-- ************************************************* -->

  <xsd:complexType name="acme-soundsType">
    <xsd:annotation>
      <xsd:documentation>
        Extension for sounds associated with a tag
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="j2ee:extensibleType">
        <xsd:sequence>
          <xsd:element name="version" type="xsd:string"/>
          <xsd:element name="tag-sound" type="xsd:string"
                       minOccurs="0" maxOccurs="unbounded"/>
          <xsd:element name="attribute-sound"
                       minOccurs="0" maxOccurs="unbounded">
            <xsd:complexType>
              <xsd:simpleContent>
                <xsd:extension base="xsd:string">
                  <xsd:attribute name="name" use="required"
                                 type="xsd:string" />
                </xsd:extension>
```

```
            </xsd:simpleContent>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="import-sound" type="xsd:string"
                     minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<!-- ********************************************** -->

</xsd:schema>
```

JSP.7.4 Validation

There are a number of reasons why the structure of a JSP page should conform to some validation rules:

- Request-time semantics; e.g. a subelement may require the information from some enclosing element at request-time .
- Authoring-tool support; e.g. a tool may require an ordering in the actions.
- Methodological constraints; e.g. a development group may want to constrain the way some features are used.

Validation can be done either at translation-time or at request-time. In general translation-time validation provides a better user experience, and the JSP 2.0 specification provides a very flexible translation-time validation mechanism.

JSP.7.4.1 Translation-Time Mechanisms

Some translation-time validation is represented in the Tag Library Descriptor. In some cases a TagExtraInfo class needs to be provided to supplement this information.

JSP.7.4.1.1 Attribute Information

The Tag Library Descriptor contains the basic syntactic information. In particular, the attributes are described including their name, whether they are optional or

mandatory, and whether they accept request-time expressions. Additionally the body-content element can be used to indicate that an action must be empty.

All constraints described in the TLD must be enforced. A tag library author can assume that the tag handler instance corresponds to an action that satisfies all constraints indicated in the TLD.

JSP.7.4.1.2 Validator Classes

A TagLibraryValidator class may be listed in the TLD for a tag library to request that a JSP page be validated. The XML view of a JSP page is exposed through a PageData class, and the validator class can do any checks the tag library author may have found appropriate.

The JSP container must uniquely identify all XML elements in the XML view of a JSP page through a jsp:id attribute. This attribute can be used to provide better information on the location of an error.

The validator class mechanism is new as of the JSP 1.2 specification. A TagLibraryValidator can be passed some initialization parameters in the TLD. This eases the reuse of validator classes. We expect that validator classes will be written based on different XML schema mechanisms (DTDs, XSchema, Relaxx, others). Standard validator classes may be incorporated into a later version of the JSP specification if a clear schema standard appears at some point.

JSP.7.4.1.3 TagExtraInfo Class Validation

Additional translation-time validation can be done using the validate method in the TagExtraInfo class. The validate method is invoked at translation-time and is passed a TagData instance as its argument. As of JSP 2.0, the default behavior of validate is to call the isValid method.

The isValid mechanism was the original validation mechanism introduced in JSP 1.1 with the rest of the Tag Extension machinery. Tag libraries that are designed to run in JSP 1.2 containers or higher should use the validator class mechanism. Tag libraries that are designed to run in JSP 2.0 containers or higher that wish to use the TagExtraInfo validation mechanism are encouraged to implement the validate method in favor of the isValid method due to the ability to provide better validation messages. Either method will work, though implementing both is not recommended.

JSP.7.4.2 Request-Time Errors

In some cases, additional request-time validation will be done dynamically within the methods in the tag handler. If an error is discovered, an instance of JspEx-

ception can be thrown. If uncaught, this object will invoke the errorpage mechanism of the JSP specification.

JSP.7.5 Conventions and Other Issues

This section is not normative, although it reflects good design practices.

JSP.7.5.1 How to Define New Implicit Objects

We advocate the following style for the introduction of implicit objects:

- Define a tag library.
- Add an action called defineObjects to define the desired objects.

The JSP page can make these objects available as follows:

```
<%@ taglib prefix="me" uri="....." %>
<me:defineObjects />
.... start using the objects....
```

This approach has the advantage of requiring no new machinery and of making very explicit the dependency.

In some cases there may be an implementation dependency in making these objects available. For example, they may be providing access to some functionality that exists only in a particular implementation. This can be done by having the tag extension class test at run-time for the existence of some implementation dependent feature and raise a run-time error (this, of course, makes the page not J2EE compliant).

This mechanism, together with the access to metadata information allows for vendors to innovate within the standard.

Note – If a feature is added to a JSP specification, and a vendor also provides that feature through its vendor-specific mechanism, the standard mechanism, as indicated in the JSP specification will “win”. This means that vendor-specific mechanisms can slowly migrate into the specification as they prove their usefulness.

JSP.7.5.2 Access to Vendor-Specific information

If a vendor wants to associate some information that is not described in the current version of the TLD with some tag library, it can do so by inserting the information in a document it controls, inserting the document in the WEB-INF portion of the Web Application where the Tag Library resides, and using the standard Servlet 2.4 mechanisms to access that information.

JSP.7.5.3 Customizing a Tag Library

A tag library can be customized at assembly and deployment time. For example, a tag library that provides access to databases may be customized with login and password information.

There is no convenient place in web.xml in the Servlet 2.4 spec for customization information A standardized mechanism is probably going to be part of a forthcoming JSP specification, but in the meantime the suggestion is that a tag library author place this information in a well-known location at some resource in the WEB-INF/ portion of the Web Application and access it via the getResource call on the ServletContext.

CHAPTER JSP.8

Tag Files

This chapter describes the details of tag files, a JSP 2.0 facility that allows page authors to author tag extensions using only JSP syntax. In the past, the ability to encapsulate presentation logic into reusable, full-featured tag libraries was only available to developers that had a reasonable amount of Java experience. Tag files bring the power of reuse to the basic page author, who are not required to know Java. When used together with JSP Fragments and Simple Tag Handlers, these concepts have the ability to simplify JSP development substantially, even for developers who do know Java.

JSP.8.1 Overview

As of JSP version 2.0, the JSP Compiler is required to recognize tag files. A tag file is a source file that provides a way for a page author to abstract a segment of JSP code and make it reusable via a custom action.

Tag files allow a JSP page author to create tag libraries using JSP syntax. This means that page authors no longer need to know Java or ask someone who knows Java to write a tag extension. Even for page authors or tag library developers who know Java, writing tag files is more convenient when developing tags that primarily output template text.

The required file extension for a tag file are .tag or .tagx. As is the case with JSP files, the actual tag may be composed of a top file that includes other files that contain either a complete tag or a segment of a tag file. Just as the recommended extension for a segment of a JSP file is .jspf, the recommended extension for a segment of a tag file is .tagf.

JSP.8.2 Syntax of Tag Files

The syntax of a tag file is similar to that of a JSP page, with the following exceptions:

- Directives - Some directives are not available or have limited availability, and some tag file specific directives are available. See Section JSP.8.5, “Tag File Directives” for a discussion on tag file directives.
- The <jsp:invoke> and <jsp:doBody> standard actions can only be used in Tag Files.

The EBNF grammar in Section JSP.1.3.10, “JSP Syntax Grammar” describes the syntax of tag files. The root production for a tag files is JSPTagDef.

See Section JSP.8.6 for details on tag files in XML syntax.

JSP.8.3 Semantics of Tag Files

For each tag file in the web application, a tag handler is made available to JSP pages and other tag files. The specifics of how this is done are left up to the Container implementation. For example, some Containers may choose to compile tag files into Java tag handlers, whereas others may decide to interpret the tag handlers.

However the Container chooses to prepare the tag handler, the following conditions must hold true for all tag handlers defined as tag files:

- The tag file implementation must keep a copy of the JspContext instance passed to it by the invoking page via the setJspContext method. This is called the *Invoking JSP Context*.
- The tag file implementation must create and maintain a second instance of JspContext called a *JSP Context Wrapper*. If the Invoking JSP Context is an instance of PageContext, the JSP Context Wrapper must also be an instance of PageContext. This wrapper must be returned when getJspContext() is called.
- For each invocation to the tag, the JSP Context Wrapper must present a clean page scope containing no initial elements. All scopes other than the page scope must be identical to those in the Invoking JSP Context and must be modified accordingly when updates are made to those scopes in the JSP Context Wrapper. Any modifications to the page scope, however, must not affect the Invoking JSP Context.

- For each attribute declared and specified, a page-scoped variable must be created in the page scope of the JSP Context Wrapper. The name of the variable must be the same as the declared attribute name. The value of the variable must be the value of the attribute passed in during invocation. For each attribute declared as optional and not specified, no page-scoped variable is created. If the tag accepts dynamic attributes, then the names and values of those dynamic attributes must be exposed to the tag file as specified in Table JSP.8-2.
- For all intents and purposes other than for synchronizing the AT_BEGIN, NESTED, and AT_END scripting variables, the effective JspContext for the tag file is the JSP Context Wrapper. For example, the jspContext scripting variable must point to the JSP Context Wrapper instead of the invoking JSP Context.
- The tag handler must behave as though a tag library descriptor entry was defined for it, in accordance with the tag, attribute, and variable directives that appear in the tag file translation unit.

It is legal for a tag file to forward to a page via the <jsp:forward> standard action. Just as for JSP pages, the forward is handled through the request dispatcher. Upon return from the RequestDispatcher.forward method, the generated tag handler must stop processing of the tag file and throw javax.servlet.jsp.SkipPageException. Similarly, if a tag file invokes a Classic Tag Handler which returns SKIP_PAGE from the doEndTag method, or if it invokes a Simple Tag Handler which throws SkipPageException in the doTag method, the generated tag handler must terminate and SkipPageException must be thrown. In either of these two cases, the doCatch and doFinally methods must be called on enclosing tags that implement the TryCatchFinally interface before returning. The doEndTag methods of enclosing classic tags must not be called.

Care should be taken when invoking a classic tag handler from a tag file. In general, SimpleTag Extensions can be used in environments other than servlet environments. However, because the Tag interface relies on PageContext, which in turn assumes a servlet environment, using classic tag handlers indirectly binds the use of the tag file to servlet environments. Nonetheless, the JSP container must allow such an invocation to occur. When a tag file attempts to invoke a classic tag handler (i.e. one that implements the Tag interface), it must cast the JspContext passed to the SimpleTag into a PageContext. In the event that the class cast fails, the invocation of the classic tag fails, and a JspException must be thrown.

JSP.8.4 Packaging Tag Files

One of the goals of tag files as a technology is to make it as easy to write a tag handler as it is to write a JSP. Traditionally, writing tag handlers has been a tedious task, with a lot of time spent compiling and packaging the tag handlers and writing a TLD to provide information to tools and page authors about the custom actions. The rules for packaging tag files are designed to make it very simple and fast to write simple tags, while still providing as much power and flexibility as classic tag handlers have.

JSP.8.4.1 Location of Tag Files

Tag extensions written in JSP using tag files can be placed in one of two locations. The first possibility is in the /META-INF/tags/ directory (or a subdirectory of /META-INF/tags/) in a JAR file installed in the /WEB-INF/lib/ directory of the web application. Tags placed here are typically part of a reusable library of tags that can be easily dropped into any web application.

The second possibility is in the /WEB-INF/tags/ directory (or a subdirectory of /WEB-INF/tags/) of the web application. Tags placed here are within easy reach and require little packaging. Only files with a .tag or .tagx extension are recognized by the container to be tag files.

Tag files that appear in any other location are not considered tag extensions and must be ignored by the JSP container. For example, a tag file that appears in the root of a web application would be treated as content to be served.

JSP.8.4.2 Packaging in a JAR

To be accessible, tag files bundled in a JAR require a Tag Library Descriptor. Tag files that appear in a JAR but are not defined in a TLD must be ignored by the JSP container.

JSP 2.0 adds an additional TLD element to describe tags within a tag library, namely <tag-file>. The <tag-file> element requires <name> and <path> subelements, which define the tag name and the full path of the tag file from the root of the JAR, respectively. In a JAR file, the <path> element must always begin with /META-INF/tags. The values for the other subelements of <tag-file> override the defaults specified in the tag directive.

Note that it is possible to combine both classic tag handlers and tag handlers implemented using tag files in the same tag library by combining the use of <tag> and <tag-file> elements under the <taglib> element. This means that in most instances the client is unaware of how the tag extension was implemented. Given

that <tag> and <tag-file> share a namespace, a tag library is considered invalid and must be rejected by the container if a <tag-file> element has a <name> subelement with the same content as a <name> subelement in a <tag> element. Any attempt to use an invalid tag library must trigger a translation error.

JSP.8.4.3 Packaging Directly in a Web Application

Tag files placed in the /WEB-INF/tags/ directory of the web application, or a subdirectory, are made easily accessible to JSPs without the need to explicitly write a Tag Library Descriptor. This makes it convenient for page authors to quickly abstract reusable JSP code by simply creating a new file and placing the code inside of it.

The JSP container must interpret the /WEB-INF/tags/ directory and each subdirectory under it, as another implicitly defined tag library containing tag handlers defined by the tag files that appear in that directory. There are no special relationships between subdirectories - they are allowed simply for organizational purposes. For example, the following web application contains three tag libraries:

```
/WEB-INF/tags/
/WEB-INF/tags/a.tag
/WEB-INF/tags/b.tag
/WEB-INF/tags/foo/
/WEB-INF/tags/foo/c.tag
/WEB-INF/tags/bar/baz/
/WEB-INF/tags/bar/baz/d.tag
```

The JSP container must generate an implicit tag library for each directory under and including /WEB-INF/tags/. This tag library can be imported only via the tagdir attribute of the taglib directive (see Section JSP.1.10.2), and has the following hard-wired values:

- <tlib-version> for the tag library defaults to 1.0
- <short-name> is derived from the directory name. If the directory is /WEB-INF/tags/, the short name is simply tags. Otherwise, the full directory path (relative to the web application) is taken, minus the /WEB-INF/tags/ prefix. Then, all / characters are replaced with -, which yields the short name. Note that short names are not guaranteed to be unique (as in /WEB-INF/tags/ versus /WEB-INF/tags/tags/ or /WEB-INF/tags/a-b/ versus /WEB-INF/tags/a/b/)
- A <tag-file> element is considered to exist for each tag file in this directory, with the following sub-elements:

- The <name> for each is the filename of the tag file, without the .tag extension.
- The <path> for each is the path of the tag file, relative to the root of the web application.

For the above example, the implicit Tag Library Descriptor for the /WEB-INF/tags/bar/baz/ directory would be:

```
<taglib>
    <tlib-version>1.0</tlib-version>
    <short-name>bar-baz</short-name>
    <tag-file>
        <name>d</name>
        <path>/WEB-INF/tags/bar/baz/d.tag</path>
    </tag-file>
</taglib>
```

Upon deployment, the JSP container must search for and process all tag files appearing in these directories and subdirectories. In processing a tag file, the container makes the custom actions defined in these tags available to JSP files.

Despite the existence of an implicit tag library, a Tag Library Descriptor in the web application can still create additional tags from the same tag files. This is accomplished by adding a <tag-file> element with a <path> that points to the tag file. In this case, the value of <path> must start with /WEB-INF/tags.

JSP.8.4.4 Packaging as Precompiled Tag Handlers

Tag files can also be compiled into Java classes and bundled as a tag library. This is useful for the situation where a tag library developer wishes to distribute a binary version of the tag library without the original source. Tag library developers that choose this form of packaging must use a tool that produces portable JSP code that uses only standard APIs. Containers are not required to provide such a tool.

JSP.8.5 Tag File Directives

This section describes the directives available within tag files, which define Simple Tag Handlers. Table JSP.8-1 outlines which directives are available in tag files:

Table JSP.8-1 *Directives available to tag files*

Directive	Available?	Interpretation/Restrictions
page	no	A tag file is not a page. The tag directive must be used instead. If this directive is used in a tag file, a translation error must result.
taglib	yes	Identical to JSP pages.
include	yes	Identical to JSP pages. Note that if the included file contains syntax unsuitable for tag files, a translation error must occur.
tag	yes	Only applicable to tag files. An attempt to use this directive in JSP pages will result in a translation error.
attribute	yes	Only applicable to tag files. An attempt to use this directive in JSP pages will result in a translation error.
variable	yes	Only applicable to tag files. An attempt to use this directive in JSP pages will result in a translation error.

JSP.8.5.1 The tag Directive

The tag directive is similar to the page directive, but applies to tag files instead of JSPs. Like the page directive, a translation unit can contain more than one instance of the tag directive, all the attributes will apply to the complete translation unit (i.e. tag directives are position independent). There shall be only one occurrence of any attribute/value defined by this directive in a given translation unit, unless the values for the duplicate attributes are identical for all occurrences. The import and pageEncoding attributes are exempt from this rule and can appear multiple times. Multiple uses of the import attribute are cumulative (with ordered set union semantics). Other such multiple attribute/value (re)definitions result in a fatal translation error if the values do not match.

The attribute/value namespace is reserved for use by this, and subsequent, JSP specification(s).

Unrecognized attributes or values result in fatal translation errors.

Examples

```
<%@ tag    display-name="Addition"
           body-content="scriptless"
           dynamic-attributes="dyn"
           small-icon="/WEB-INF/sample-small.jpg"
           large-icon="/WEB-INF/sample-large.jpg"
           description="Sample usage of tag directive" %>
```

Syntax

<%@ tag *tag_directive_attr_list* %>

tag_directive_attr_list ::=

```
{ display-name="display-name"    }
{ body-content="scriptless|tagdependent|empty" }
{ dynamic-attributes="name"      }
{ small-icon="small-icon"        }
{ large-icon="large-icon"        }
{ description="description"      }
{ example="example"              }
{ language="scriptingLanguage"   }
{ import="importList"            }
{ pageEncoding="peinfo"          }
{ isELIgnored="true|false"       }
```

The details of the attributes are as follows:

Table JSP.8-2 *Details of tag directive attributes*

display-name	(optional) A short name that is intended to be displayed by tools. Defaults to the name of the tag file, without the .tag extension.
body-content	(optional) Provides information on the content of the body of this tag. Can be either empty, tagdependent, or scriptless. A translation error will result if JSP or any other value is used. Defaults to scriptless.

Table JSP.8-2 *Details of tag directive attributes*

dynamic-attributes	(optional) The presence of this attribute indicates this tag supports additional attributes with dynamic names. If present, the generated tag handler must implement the javax.servlet.jsp.tagext.DynamicAttributes interface, and the container must treat the tag as if its corresponding TLD entry contained <dynamic-attributes>true</dynamic-attributes>. The implementation must not reject any attribute names. The value identifies a page scoped attribute in which to place a Map containing the names and values of the dynamic attributes passed during this invocation. The Map must contain each dynamic attribute name as the key and the dynamic attribute value as the corresponding value. Only dynamic attributes with no uri are to be present in the Map; all other dynamic attributes are ignored. A translation error will result if there is a tag directive with a dynamic-attributes attribute equal to the value of a name-given attribute of a variable directive or equal to the value of a name attribute of an attribute directive in this translation unit.
small-icon	(optional) Either a context-relative path, or a path relative to the tag source file, of an image file containing a small icon that can be used by tools. Defaults to no small icon.
large-icon	(optional) Either a context-relative path, or a path relative to the tag source file, of an image file containing a large icon that can be used by tools. Defaults to no large icon.
description	(optional) Defines an arbitrary string that describes this tag. Defaults to no description.
example	(optional) Defines an arbitrary string that presents an informal description of an example of a use of this action. Defaults to no example.
language	(optional) Carries the same syntax and semantics of the language attribute of the page directive.
import	(optional) Carries the same syntax and semantics of the import attribute of the page directive.

Table JSP.8-2 *Details of tag directive attributes*

pageEncoding	(optional) Carries the same syntax and semantics of the pageEncoding attribute in the page directive. However, there is no corresponding global configuration element in web.xml. The pageEncoding attribute cannot be used in tag files in XML syntax.
isELIgnored	(optional) Carries the same syntax and semantics of the isELIgnored attribute of the page directive. However, there is no corresponding global configuration element in web.xml.

JSP.8.5.2 The attribute Directive

The attribute directive is analogous to the <attribute> element in the Tag Library Descriptor, and allows for the declaration of custom action attributes.

Examples

```
<%@ attribute   name="x" required="true" fragment="false"
                rtexprvalue="false" type="java.lang.Integer"
                description="The first operand" %>

<%@ attribute   name="y" type="java.lang.Integer" %>

<%@ attribute   name="prompt" fragment="true" %>
```

Syntax

<%@ attribute *attribute_directive_attr_list* %>

attribute_directive_attr_list ::=

```
                name="attribute-name"
                { required="true|false"          }
                { fragment="true|false"          }
                { rtexprvalue="true|false"       }
                { type="type"                    }
                { description="description"      }
```

The details of the attributes are as follows:

Table JSP.8-3 *Details of attribute directive attributes*

name	(required) The unique name of the attribute being declared. A translation error must result if more than one attribute directive appears in the same translation unit with the same name. A translation error will result if there is an attribute directive with a name attribute equal to the value of the name-given attribute of a variable directive or the dynamic-attributes attribute of a tag directive in this translation unit.
required	(optional) Whether this attribute is required (true) or optional (false). Defaults to false if not specified.
fragment	(optional) Whether this attribute is a fragment to be evaluated by the tag handler (true) or a normal attribute to be evaluated by the container prior to being passed to the tag handler. If this attribute is true, the type attribute is fixed at javax.servlet.jsp.tagext.JspFragment and a translation error will result if the type attribute is specified. Also, if this attribute is true, the rtexprvalue attribute is fixed at true and a translation error will result if the rtexprvalue attribute is specified. Defaults to false.
rtexprvalue	(optional) Whether the attribute's value may be dynamically calculated at runtime by a scriptlet expression. Unlike the corresponding TLD element, this attribute defaults to true.
type	(optional) The runtime type of the attribute's value. Defaults to java.lang.String if not specified. It is a translation error to specify a primitive type.
description	(optional) Description of the attribute. Defaults to no description.

JSP.8.5.3 The variable Directive

The variable directive is analogous to the <variable> element in the Tag Library descriptor, and defines the details of a variable exposed by the tag handler to the calling page.

See Section JSP.7.1.4.7, "Actions Defining Scripting Variables" for more details.

Examples

```
<%@ variable   name-given="sum"
               variable-class="java.lang.Integer"
               scope="NESTED"
               declare="true"
               description="The sum of the two operands" %>

<%@ variable   name-given="op1"
               variable-class="java.lang.Integer"
               description="The first operand" %>

<%@ variable   name-from-attribute="var" alias="result" %>
```

Syntax

<%@ variable *variable_directive_attr_list* %>

variable_directive_attr_list ::=

```
                    (       name-given="output-name"
                      |     (   name-from-attribute="attr-name"
                                alias="local-name"
                            )
                    )
                    { variable-class="output-type"              }
                    { declare="true|false"                      }
                    { scope="AT_BEGIN|AT_END|NESTED"            }
                    { description="description"                 }
```

The details of the attributes are as follows:

Table JSP.8-4 *Details of variable directive attributes*

name-given	Defines a scripting variable to be defined in the page invoking this tag. Either the name-given attribute or the name-from-attribute attribute must be specified. Specifying neither or both will result in a translation error. A translation error will result if two variable directives have the same name-given. A translation error will result if there is a variable directive with a name-given attribute equal to the value of the name attribute of an attribute directive or the dynamic-attributes attribute of a tag directive in this translation unit.

Table JSP.8-4 *Details of variable directive attributes*

name-from-attribute	Defines a scripting variable to be defined in the page invoking this tag. The specified name is the name of an attribute whose (translation-time) value at of the start of the tag invocation will give the name of the variable. A translation error will result if there is no attribute directive with a name attribute equal to the value of this attribute that is of type java.lang.String, is required and not an rtexprvalue. Either the name-given attribute or the name-from-attribute attribute must be specified. Specifying neither or both will result in a translation error. A translation error will result if two variable directives have the same name-from-attribute.
alias	Defines a locally scoped attribute to hold the value of this variable. The container will synchronize this value with the variable whose name is given in name-from-attribute. Required when name-from-attribute is specified. A translation error must occur if used without name-from-attribute. A translation error must occur if the value of alias is the same as the value of a name attribute of an attribute directive or the name-given attribute of a variable directive in the same translation unit.
variable-class	(optional) The name of the class of the variable. The default is java.lang.String.
declare	(optional) Whether the variable is declared or not in the calling page/tag file, after this tag invocation. true is the default.
scope	(optional) The scope of the scripting variable defined. Can be either AT_BEGIN, AT_END, or NESTED. Defaults to NESTED.
description	(optional) An optional description of this variable. Defaults to no description.

JSP.8.6 Tag Files in XML Syntax

Tag files can be authored using the XML syntax, as described in JSP Documents, Chapter JSP.6. This section describes the few distinctions from the case of JSP documents.

Tag files in XML syntax must have the extension .tagx. All files with extension .tagx according to the rules in Section JSP.8.4.1 are tag files in XML syntax. Conversely, files with extension .tag are not in XML syntax.

The jsp:root element can, but needs not, appear in tag files in XML syntax. A jsp:root element cannot appear in a tag file in JSP syntax.

As indicated in Section JSP.5.16, "<jsp:output>", the default for tag files, in either syntax, is not to generate the xml declaration. The element jsp:output can be used to change that default for tag files in XML syntax.

Finally, the tag directive in a tag file in XML syntax cannot include a pageEncoding attribute; the encoding is inferred using the conventions for XML documents. Using the pageEncoding attribute shall result in a translation-time error.

JSP.8.7 XML View of a Tag File

Similar to JSP pages, tag files have an equivalent XML document, the XML view of a tag file, that is exposed to the translation phase for validation. During the translation phase for a tag file, a tag XML view is created and passed to all TLVs declared in all tag libraries declared in the tag file.

The XML view of a tag file is identical to the XML view of a JSP, except that there are additional XML elements defined to handle tag file specific features. The XML view of a tag file is obtained in the same way that the XML view of a JSP page is obtained (see Chapter JSP.10).

JSP.8.8 Implicit Objects

Tag library developers writing tag files have access to certain implicit objects that are always available for use within scriptlets and expressions through scripting variables that are declared implicitly at the beginning of the tag handler implementation. All scripting languages are required to provide access to these objects.

Each implicit object has a class or interface type defined in a core Java technology or Java Servlet API package, as shown in Table JSP.8-5.

Table JSP.8-5 *Implicit Objects Available in Tag Files*

Variable Name	Type	Semantics & Scope
request	protocol dependent subtype of: javax.servlet.ServletRequest e.g: javax.servlet.http.HttpServletRequest	The request triggering the service invocation. request scope.
response	protocol dependent subtype of: javax.servlet.ServletResponse, e.g: javax.servlet.http.HttpServletResponse	The response to the request. page scope.
jspContext	javax.servlet.jsp.JspContext	The JspContext for this tag file. page scope.
session	javax.servlet.http.HttpSession	The session object created for the requesting client (if any). This variable is only valid for HTTP protocols. session scope
application	javax.servlet.ServletContext	The servlet context obtained from the servlet configuration object (as in the call getServlet-Config(). getContext()) application scope
out	javax.servlet.jsp.JspWriter	An object that writes into the output stream. page scope
config	javax.servlet.ServletConfig	The ServletConfig for this JSP page page scope

Object names with prefixes jsp, _jsp, jspx and _jspx, in any combination of upper and lower case, are reserved by the JSP specification.

JSP.8.9 Variable Synchronization

Just as is the case for all tag handlers, a tag file is able to communicate with its calling page via variables. As mentioned earlier, in tag files, variables are declared using the variable directive. Though the scopes of variables are similar to those in classic tag handlers, the semantics are slightly different. The intent is to be able to emulate IN and OUT parameters using attributes and variables, which appear as page-scoped attributes local to the tag file, and are synchronized with the calling page's JspContext at various points.

The name-from-attribute and alias attributes of the variable directive can be used to allow the caller to customize the name of the variable in the calling page while referring to a constant name in the tag file. When using these attributes, the name of the variable in the calling page is derived from the value of name-from-attribute at the time the tag was called. The name of the corresponding variable in the tag file is the value of alias.

- IN parameters - Use attributes. For each attribute, a page-scoped attribute is made available in the JspContext of the tag file. The page-scoped attribute is initialized to the value of the attribute when the tag is called. No further synchronization is performed.
- OUT parameters - Use variables with scope AT_BEGIN or AT_END. For each AT_BEGIN or AT_END variable, a page-scoped attribute is made available in the JspContext of the tag file. The scoped attribute is not initialized. Synchronization is performed at the end of the tag for AT_BEGIN and AT_END and also before the invocation of a fragment for AT_BEGIN. See Table JSP.8-6 for details.
- Nested parameters - Use variables with scope AT_BEGIN or NESTED. For each AT_BEGIN or NESTED variable, a page-scoped attribute is made available in the JspContext of the tag file. The scoped attribute is not initialized. Synchronization is performed before each fragment invocation for AT_BEGIN and NESTED, and also after the end of the tag for AT_BEGIN. See Table JSP.8-6 for details.

JSP.8.9.1 Synchronization Points

The JSP container is required to generate code to handle the synchronization of each declared variable. The details of how and when each variable is synchronized varies by the variable's scope, as per Table JSP.8-6.

Table JSP.8-6 *Variable synchronization behavior*

	AT_BEGIN	NESTED	AT_END
Beginning of tag file	do nothing	save	do nothing
Before any fragment	tag → page	tag → page	do nothing
After any fragment	do nothing	do nothing	do nothing
End of tag file	tag → page	restore	tag → page

The following list describes what each synchronization action means. If name-given is used, the name of the variable in the calling page (referred to as *P*) and the name of the variable in the tag file (referred to as *T*) are the same and are equal to the value of name-given. If name-from-attribute is used, the name of *P* is equal to the value of the attribute (at the time the page was called) specified by the value of name-from-attribute and the name of *T* is equal to the value of the alias attribute.

- tag → page - For this variable, if *T* exists in the tag file, create/update *P* in the calling page. If a *T* does not exist in the tag file, and *P* does exist in the calling page, *P* is removed from the calling page's page scope. If the declare attribute for this variable is set to true, a corresponding scripting variable is declared in the calling page or tag file, as with any other tag handler. If this scripting variable would not be accessible in the context in which it is defined, the container need not declare the scripting variable (for example in a scriptless body).
- save - For this variable, save the value of *P*, for later restoration. If *P* did not exist, remember that fact.
- restore - For this variable, restore the value of *P* in the calling page, from the value saved earlier. If *P* did not exist before, ensure it does not exist now.

All variable synchronization and restoration that occurs at the end of a tag file must occur regardless of whether an exception is thrown inside the tag file. All variable synchronization that occurs after the invocation of a fragment must occur regardless of whether an exception occured while invoking the fragment.

JSP.8.9.2 Synchronization Examples

The following examples help illustrate how variable synchronization works between a tag file and its calling page.

JSP.8.9.2.1 Example of AT_BEGIN

In this example, the AT_BEGIN scope is used to pass a variable to the tag's body, and make it available to the calling page at the end of the tag invocation.

```
<%-- page.jsp --%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="my" tagdir="/WEB-INF/tags" %>
<c:set var="x" value="1"/>
${x} <%-- (x == 1) --%>
<my:example>
    ${x} <%-- (x == 2) --%>
    <c:set var="x" value="3"/>
</my:example>
${x} <%-- (x == 4) --%>

<%-- /WEB-INF/tags/example.tag --%>
<%@ variable name-given="x" scope="AT_BEGIN" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
${x} <%-- (x == null) --%>
<c:set var="x" value="2"/>
<jsp:doBody/>
${x} <%-- (x == 2) --%>
<c:set var="x" value="4"/>
```

JSP.8.9.2.2 Example of AT_BEGIN and name-from-attribute

Like the previous example, in this example the AT_BEGIN scope is used to pass a variable to the tag's body, and make it available to the calling page at the end of the tag invocation. The name of the attribute is customized via name-from-attribute.

```
<%-- page.jsp --%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="my" tagdir="/WEB-INF/tags" %>
<c:set var="x" value="1"/>
${x} <%-- (x == 1) --%>
<my:example var="x">
    ${x} <%-- (x == 2) --%>
    ${result} <%-- (result == null) --%>
    <c:set var="x" value="3"/>
    <c:set var="result" value="invisible"/>
</my:example>
${x} <%-- (x == 4) --%>
${result} <%-- (result == 'invisible') --%>

<%-- /WEB-INF/tags/example.tag --%>
<%@ attribute name="var" required="true" rtexprvalue="false"%>
<%@ variable alias="result" name-from-attribute="var" scope="AT_BEGIN" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
${x} <%-- (x == null) --%>
${result} <%-- (result == null) --%>
<c:set var="x" value="ignored"/>
<c:set var="result" value="2"/>
<jsp:doBody/>
${x} <%-- (x == 'ignored') --%>
${result} <%-- (result == 2) --%>
<c:set var="x" value="still_ignored"/>
<c:set var="result" value="4"/>
```

JSP.8.9.2.3 Example of NESTED

In this example, the NESTED scope is used to make a private variable available to the calling page. The original value is restored when the tag is done.

```
<%-- page.jsp --%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="my" tagdir="/WEB-INF/tags" %>
<c:set var="x" value="1"/>
${x} <%-- (x == 1) --%>
<my:example>
    ${x} <%-- (x == 2) --%>
    <c:set var="x" value="3"/>
</my:example>
${x} <%-- (x == 1) --%>
```

```
<%-- /WEB-INF/tags/example.tag --%>
<%@ variable name-given="x" scope="NESTED" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
${x} <%-- (x == null) --%>
<c:set var="x" value="2"/>
<jsp:doBody/>
${x} <%-- (x == 2) --%>
<c:set var="x" value="4"/>
```

JSP.8.9.2.4 Example of AT_END

In this example, the AT_END scope is used to return a value to the page. The body of the tag is not affected.

```
<%-- page.jsp --%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="my" tagdir="/WEB-INF/tags" %>
<c:set var="x" value="1"/>
${x} <%-- (x == 1) --%>
<my:example>
    ${x} <%-- (x == 1) --%>
    <c:set var="x" value="3"/>
</my:example>
${x} <%-- (x == 4) --%>

<%-- /WEB-INF/tags/example.tag --%>
<%@ variable name-given="x" scope="AT_END" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
${x} <%-- (x == null) --%>
<c:set var="x" value="2"/>
<jsp:doBody/>
${x} <%-- (x == 2) --%>
<c:set var="x" value="4"/>
```

JSP.8.9.2.5 Example of Removing Parameters

This example illustrates how the tag file can remove objects from the page scope of the calling page during synchronization.

```
<%-- page.jsp --%>
<%@ taglib prefix=”my” tagdir=”/WEB-INF/tags” %>
<%@ taglib prefix=”c” uri=”http://java.sun.com/jsp/jstl/core” %>
<c:set var=”x” value=”2”/>
${x}
<my:tag1>
     ‘${x}’
</my:tag1>
${x}

<%-- /WEB-INF/tags/example.tag --%>
<%@ variable name-given=”x” scope=”NESTED” %>
<%@ taglib prefix=”c” uri=”http://java.sun.com/jsp/jstl/core” %>
<c:set var=”x” value=”1”/>
<jsp:doBody/>
<c:remove var=”x”/>
<jsp:doBody/>
```

The expected output of this example is: 2 ‘1’ ‘’ 2

CHAPTER JSP.9

Scripting

This chapter describes the details of the Scripting Elements when the language directive value is java.

The scripting language is based on the Java programming language (as specified by “The Java Language Specification”), but note that there is no valid JSP page, or a subset of a page, that is a valid Java program.

The following sections describe the details of the relationship between the scripting declarations, scriptlets, and scripting expressions, and the Java programming language. The description is in terms of the structure of the JSP page implementation class. A JSP Container need not generate the JSP page implementation class, but it must behave as if one exists.

JSP.9.1 Overall Structure

Some details of what makes a JSP page legal are very specific to the scripting language used in the page. This is especially complex since scriptlets are language fragments, not complete language statements.

JSP.9.1.1 Valid JSP Page

A JSP page is valid for a Java Platform if and only if the JSP page implementation class defined by Table JSP.9-1 (after applying all include directives), together with any other classes defined by the JSP container, is a valid program for the given Java Platform, and if it passes the validation methods for all the tag libraries associated with the JSP page.

JSP.9.1.2 Reserved Names

Sun Microsystems reserves all names of the form {_}jsp_* and {_}jspx_*, in any combination of upper and lower case, for the JSP specification. Names of this form that are not defined in this specification are reserved by Sun for future expansion.

JSP.9.1.3 Implementation Flexibility

The transformations described in this chapter need not be performed literally. An implementation may implement things differently to provide better performance, lower memory footprint, or other implementation attributes.

Table JSP.9-1 *Structure of the JavaProgramming Language Class*

Optional imports clause as indicated via jsp directive	import name1
SuperClass is either selected by the JSP container or by the JSP author via the jsp directive. Name of class (_jspXXX) is implementation dependent.	class _jspXXX extends SuperClass
Start of the body of a JSP page implementation class	{
(1) Declaration Section	// declarations...
signature for generated method	public void _jspService(<ServletRequestSubtype> request, <ServletResponseSubtype> response) throws ServletException, IOException {

Table JSP.9-1 *Structure of the JavaProgramming Language Class*

(2) Implicit Objects Section	// code that defines and initializes request, response, page, pageContext etc.
(3) Main Section	// code that defines request/response mapping
close of _jspService method	}
close of _jspXXX	}

JSP.9.2 Declarations Section

The declarations section corresponds to the declaration elements.

The contents of this section is determined by concatenating all the declarations in the page in the order in which they appear.

JSP.9.3 Initialization Section

This section defines and initializes the implicit objects available to the JSP page. See Section JSP.1.8.3, “Implicit Objects”.

JSP.9.4 Main Section

This section provides the main mapping between a request and a response object.

The content of code segment 2 is determined from scriptlets, expressions, and the text body of the JSP page. The elements are processed sequentially in the order in which they appear in the page. The translation for each one is determined as indicated below, and its translation is inserted into this section. The translation depends on the element type:

JSP.9.4.1 Template Data

Template data is transformed into code that will place the template data into the stream named by the implicit variable out when the code is executed. White space is preserved.

Ignoring quotation issues and performance issues, this corresponds to a statement of the form:

Original	Equivalent Text
template	out.print(template)

JSP.9.4.2 Scriptlets

A *scriptlet* is transformed into its code fragment.:

Original	Equivalent Text
<% fragment %>	fragment

JSP.9.4.3 Expressions

An *expression* is transformed into a Java statement to insert the value of the expression, converted to java.lang.String if needed, into the stream named by the implicit variable out. No additional newlines or space is included.

Ignoring quotation and performance issues, this corresponds to a statement of the form:

Original	Equivalent Text
<%= expression %>	out.print(expression)

JSP.9.4.4 Actions

An action defining one or more objects is transformed into one or more variable declarations for those objects, together with code that initializes the variables. Their visibility is affected by other constructs, for example scriptlets.

The semantics of the action type determines the names of the variables (usually the name of an id attribute, if present) and their type. The only standard action in the JSP specification that defines objects is the jsp:useBean action. The name of the variable introduced is the name of the id attribute and its type is the type of the class attribute.

Original	Equivalent Text

<x:tag>
  foo
</x:tag>
	declare AT_BEGIN variables
	{
	  declare NESTED variables
	  transformation of foo
	}
	declare AT_END variables

Note that the value of the scope attribute does not affect the visibility of the variables within the generated program. It affects where and thus for how long there will be additional references to the object denoted by the variable.

CHAPTER JSP.10

XML View

This chapter provides details on the XML view of a JSP page and tag files. The XML views are used to enable validation of JSP pages and tag files..

JSP.10.1 XML View of a JSP Document, JSP Page or Tag File

This section describes the XML view of a JSP page or tag file: the mapping between a JSP page, JSP document or tag file, and an XML document describing it.

JSP.10.1.1 JSP Documents and Tag Files in XML Syntax

The XML view of a JSP document or of a tag file written in XML syntax is very close to the original JSP page. Only five transformations are performed:

- Expand all include directives into the JSP content they include. See Section JSP.1.10.5 for the semantics of mixing XML and standard syntax content.
- Add a jsp:root element as the root element if the JSP document or tag file in XML syntax does not have it.
- Set the value of the pageEncoding attribute of the page directive to "UTF-8". The page directive and the pageEncoding attribute are added if they don't exist already.
- Set the value of the contentType attribute of the page directive to the value that the container will pass to ServletResponse.setContentType(), determined as described in Section JSP.4.2, "Response Character Encoding". The page directive and the contentType attribute are added if they don't exist already.
- Add the jsp:id attribute (see Section JSP.10.1.13).

JSP.10.1.2 JSP Pages or Tag Files in JSP Syntax

The XML view of a JSP page or tag file written in standard syntax is defined by the following transformation:

- Expand all include directives into the JSP content they include. See Section JSP.1.10.5 for the semantics of mixing XML and standard syntax content.
- Add a jsp:root element as the root, with appropriate xmlns:jsp attribute, and convert the taglib directive into xmlns: attributes of the jsp:root element.
- Convert declarations, scriptlets, and expressions into valid XML elements as described in Section JSP.6.3.2 and the following sections.
- Convert request-time attribute expressions as in Section JSP.10.1.11.
- Convert JSP quotations to XML quotations.
- Create jsp:text elements for all template text.
- Add the jsp:id attribute (see Section JSP.10.1.13).

Note that the XML view of a JSP page or tag file has no DOCTYPE information; see Section JSP.10.2.

A quick overview of the transformation is shown in Table JSP.10-1:

Table JSP.10-1 *XML View Transformations*

JSP element	XML view
<%-- comment --%>	removed
<%@ page ... %>	<jsp:directive.page ... />. Add jsp:id
<%@ taglib ... %>	jsp:root element is annotated with namespace information. Add jsp:id.
<%@ include ... %>	expanded in place
<%! ... %>	<jsp:declaration> ... </jsp:declaration>. Add jsp:id.
<% ... %>	<jsp:scriptlet> ... </jsp:scriptlet>. Add jsp:id.
<%= ... %>	<jsp:expression> ... </jsp:expression>. Add jsp:id.
Standard action	Replace with XML syntax (adjust request-time expressions; add jsp:id)

Table JSP.10-1 *XML View Transformations*

JSP element	XML view
Custom action	As is (adjust request-time expressions; add jsp:id)
template	Replace with jsp:text. Add jsp:id.
<%@ tag ... %>	<jsp:directive.tag ... />. Add jsp:id. [tag files only]
<%@ attribute ... %>	<jsp:directive.attribute ... />. Add jsp:id. [tag files only]
<%@ variable ... %>	<jsp:directive.variable ... />. Add jsp:id. [tag files only]

In more detail:

JSP.10.1.3 JSP Comments

JSP comments (of the form <%-- comment --%>) are not passed through to the XML view of a JSP page.

JSP.10.1.4 The page Directive

A page directive of the form:

<%@ page { attr="value" }* %>

is translated into an element of the form:

<jsp:directive.page { attr="value" }* />

The value of the pageEncoding attribute is set to "UTF-8". The value of the contentType attribute is set to the value that the container will pass to ServletResponse.setContentType(), determined as described in Section JSP.4.2, "Response Character Encoding". The page directive and both attributes are added if they don't exist already.

JSP.10.1.5 The taglib Directive

A taglib directive of the form

<%@ taglib uri="*uriValue*" prefix="*prefix*" %>

is translated into an xmlns:prefix attribute on the root of the JSP document, with a value that depends on uriValue. If uriValue is a relative path, then the value used is urn:jsptld:*uriValue*; otherwise, the uriValue is used directly.

A taglib directive of the form

<%@ taglib tagdir="*tagDirValue*" prefix="*prefix*" %>

is translated into an xmlns:prefix attribute on the root of the JSP document, with a value of the form urn:jsptagdir:*tagDirValue*.

JSP.10.1.6 The include Directive

An include directive of the form

<%@ include file="*value*" %>

is expanded into the JSP content indicated by value. This is done to allow for validation of the page.

JSP.10.1.7 Declarations

Declarations are translated into a jsp:declaration element. For example, the second example from Section JSP.1.12.1:

<%! public String f(int i) { if (i<3) return("..."); ... } %>

is translated into the following.

<jsp:declaration> <![CDATA[public String f(int i) { if (i<3) return("..."); }]]> </jsp:declaration>

Alternatively, we could use an < and instead say:

<jsp:declaration> public String f(int i) { if (i<3) return("..."); } </jsp:declaration>

JSP.10.1.8 Scriptlets

Scriptlets are translated into a jsp:scriptlet element. In the XML document corresponding to JSP pages, directives are represented using the syntax:

<jsp:scriptlet> code fragment goes here </jsp:scriptlet>

JSP.10.1.9 Expressions

In the XML document corresponding to JSP pages, directives are represented using the jsp:expression element:

```
<jsp:expression> expression goes here </jsp:expression>
```

JSP.10.1.10 Standard and Custom Actions

The syntax for both standard and action elements is based on XML. The transformations needed are due to quoting conventions and the syntax of request-time attribute expressions.

JSP.10.1.11 Request-Time Attribute Expressions

Request-time attribute expressions are of the form <%= expression %>. Although this syntax is consistent with the syntax used elsewhere in a JSP page, it is not a legal XML syntax. The XML mapping for these expressions is into values of the form %= expression %, where the JSP specification quoting convention has been converted to the XML quoting convention.

Request-time attribute values can also be specified using EL expressions of the form ${expression}. Expressions of this form are represented verbatim in the XML view.

The XML view of an escaped EL expression can be obtained as follows:

- The XML view of an unescaped expression ${foo} is ${foo}.
- The XML view of an escaped expression \${foo} is \${foo}.
- For each escaped \ preceeding an unescaped expression ${foo}, a ${'\\'} must be generated in the XML view, and neighboring generated ${'\\'} expressions must be combined.

Table JSP.10-2 illustrates these rules. Assume the EL expression ${foo} evaluates to [bar] and that EL is enabled for this translation unit.

Table JSP.10-2 *XML View of an Escaped EL Expression in a Request-time Attribute Value*

Attribute Value	XML View	Result
${foo}	${foo}	[bar]
\${foo}	\${foo}	${foo}

Table JSP.10-2 *XML View of an Escaped EL Expression in a Request-time Attribute Value*

Attribute Value	XML View	Result
\\${foo}	${'\\'}${foo}	\[bar]
\\\${foo}	\\${foo}	\${foo}
\\\\${foo}	${'\\\\'}${foo}	\\[bar]
\\\\\${foo}	\\\${foo}	\\${foo}
\\\\\\${foo}	${'\\\\\\'}${foo}	\\\[bar]
...	...	...

JSP.10.1.12 Template Text and XML Elements

All text that is uninterpreted by the JSP translator is converted into the body for a jsp:text element. As a consequence no XML elements of the form described in Section JSP.6.3.9, "Template Content" will appear in the XML view of a JSP page written in JSP syntax.

Because \\ is not an escape sequence within template text in the standard syntax, no special transformation needs to be done to obtain the XML view of an escaped EL expression that appears in template text.

Table JSP.10-3 illustrates how the XML view of an escaped EL expression is obtained. Assume the EL expression ${foo} evaluates to [bar] and that EL is enabled for this translation unit.

Table JSP.10-3 *XML View of an Escaped EL Expression in Template Text*

Attribute Value	XML View	Result
${foo}	${foo}	[bar]
\${foo}	\${foo}	${foo}
\\${foo}	\\${foo}	\${foo}
\\\${foo}	\\\${foo}	\\${foo}
...	...	...

JSP.10.1.13 The jsp:id Attribute

A JSP container must support a jsp:id attribute. This attribute can only be present in the XML view of a JSP page and can be used to improve the quality of translation time error messages.

The XML view of any JSP page will have an additional jsp:id attribute added to all XML elements. This attribute is given a value that is unique over all elements in the XML view. The prefix for the id attribute need not be "jsp" but it must map to the namespace http://java.sun.com/JSP/Page. In the case where the page author has redefined the jsp prefix, an alternative prefix must be used by the container. See Chapter JSP.13 for more details.

JSP.10.1.14 The tag Directive

The tag directive is applicable to tag files only. A tag directive of the form:

```
<%@ tag { attr=”value” }* %>
```

is translated into an element of the form:

```
<jsp:directive.tag { attr=”value” }* />
```

The value of the pageEncoding attribute is set to "UTF-8". A tag directive and the pageEncoding attribute are added if they don’t exist already.

JSP.10.1.15 The attribute Directive

The attribute directive is applicable to tag files only. An attribute directive of the form:

```
<%@ attribute { attr=”value” }* %>
```

is translated into an element of the form:

```
<jsp:directive.attribute { attr=”value” }* />
```

JSP.10.1.16 The variable Directive

The variable directive is applicable to tag files only. A variable directive of the form:

```
<%@ variable { attr=”value” }* %>
```

is translated into an element of the form:

```
<jsp:directive.variable { attr="value" }* />
```

JSP.10.2 Validating an XML View of a JSP page

The XML view of a JSP page is a namespace-aware document and it cannot be validated against a DTD except in the most simple cases. To reduce confusion and possible unintended performance consequences, the XML view of a JSP page will not include a DOCTYPE.

There are several mechanisms that are aware of namespaces that can be used to do validation of XML views of JSP pages. The most popular mechanism is the W3C XML Schema language, but others are also suited, including some very simple ones that may check, for example, that only some elements are being used, or, inversely, that they are not used. The TagLibraryValidator for a tag library permits encapsulating this knowledge with a tag library.

The TagLibraryValidator acts on the XML view of the JSP page. If the page was authored in JSP syntax, that view does not provide any detail on template data (all being grouped inside jsp:text elements), but fine detail can be described when using JSP documents[1].

JSP.10.3 Examples

This section presents various examples of XML Views. The first shows a JSP page in XML syntax that includes XML fragments. The second shows a JSP page in JSP syntax and its mapping to XML syntax. The three following examples illustrate the semantics of cross-syntax translation-time includes and the effect on the XML View.

JSP.10.3.1 A JSP document

This is an example of a very simple JSP document that has some template XML elements. This particular example describes a table that is a collection of 3 rows, with numeric values 1, 2, 3. The JSP Standard Tag Library is being used:

---

[1] Similarly, when applying an XSLT transformation to a JSP document, XML fragments will be plainly visible, while the content of jsp:text elements will not

```
<?xml version="1.0"?>
<table>
    <c:forEach
        xmlns:c="http://java.sun.com/jsp/jstl/core"
        var="counter" begin="1" end="3">
            <row>${counter}</row>
    </c:forEach>
</table>
```

JSP.10.3.2 A JSP page and its corresponding XML View

Here is an example of mapping between JSP and XML syntax.
For this JSP page:

```
<html>
<title>positiveTagLib</title>
<body>

<%@ taglib uri="http://java.apache.org/tomcat/examples-taglib" prefix="eg" %>
<%@ taglib uri="/tomcat/taglib" prefix="test" %>
<%@ taglib uri="WEB-INF/tlds/my.tld" prefix="temp" %>

<eg:test toBrowser="true" att1="Working">
Positive Test taglib directive </eg:test>
</body>
</html>
```

The XML View of the previous page is:

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
    xmlns:eg="http://java.apache.org/tomcat/examples-taglib"
    xmlns:test="urn:jsptld:/tomcat/taglib"
    xmlns:temp="urn:jsptld:/WEB-INF/tlds/my.tld"
    version="2.0">

<jsp:text><![CDATA[<html>
<title>positiveTagLib</title>
<body>




]]></jsp:text>
<eg:test toBrowser="true" att1="Working">
<jsp:text>Positive test taglib directive</jsp:text>
</eg:test>
<jsp:text><![CDATA[
</body>
</html>
]]></jsp:text>
</jsp:root>
```

JSP.10.3.3 Clearing Out Default Namespace on Include

This example illustrates the need to clear out the default namespace when doing a translation-time include of a JSP document:

```
<!-- a.jspx -->
<elementA>
    <tagB xmlns="http://namespace1">
        <jsp:directive.include file="b.jspx" />
    </tagB>
</elementA>

<!-- b.jspx -->
<elementC />
```

The resulting XML View for these two JSP documents is:

```
<jsp:root>
    <elementA>
        <tagB xmlns="http://namespace1">
            <elementC />
        </tagB>
    </elementA>
</jsp:root>
```

JSP.10.3.4 Taglib Direcive Adds to Global Namespace

This example illustrates the effect of the taglib directive on the XML View. Notice how the taglib directive always affects the <jsp:root> element, independent of where it is encountered.

```
<!-- c.jspx -->
<elementD>
    <jsp:directive.include file="d.jsp" />
    <jsp:directive.include file="e.jsp" />
</elementD>

<%-- d.jsp --%>
<%@ taglib prefix="x" uri="http://namespace2" %>
<x:tagE />

<%-- e.jsp --%>
<x:tagE />
```

The resulting XML View of these documents and pages is:

```
<jsp:root xmlns:x="http://namespace2">
    <elementD>
        <x:tagE />
        <x:tagE />
    </elementD>
</jsp:root>
```

JSP.10.3.5 Collective Application of Inclusion Semantics

This example illustrates how the various translation-time include semantics are collectively applied:

```
<%-- f.jsp --%>
<%@ taglib prefix="m" uri="http://namespace3" %>
<%@ include file="g.jspx" %>

<!-- g.jspx -->
<tagF xmlns="http://namespace4" />
    <y:tagG xmlns:y="http://namespace5">
        <tagH />
        <jsp:directive.include file="i.jspx" />
    </y:tagG>
    <jsp:directive.include file="h.jsp" />
    <tagI />
</tagF>

<%-- h.jsp --%>
<%@ taglib prefix="n" uri="http://namespace6" %>
<m:tagJ />
<n:tagK />

<!-- i.jspx -->
<jsp:root>
    <y:tagL xmlns:y="http://namespace7">
        <elementM />
        <jsp:directive.include file="h.jsp" />
    </y:tagL>
</jsp:root>
```

The resulting XML View of these documents and pages is:

```
<jsp:root      xmlns:m="http://namespace3"
               xmlns:n="http://namespace6">
    <tagF xmlns="http://namespace4">
        <y:tagG xmlns:y="http://namespace5">
            <tagH />
            <y:tagL xmlns="" xmlns:y="http://namespace7">
                <elementM />
                <m:tagJ />
                <n:tagK />
            </y:tagL>
        </y:tagG>
        <m:tagJ />
        <n:tagK />
        <tagI />
    </tagF>
</jsp:root>
```

Part II

The next chapters provide detail specification information on some portions of the JSP specification that are intended for JSP Container Vendors, JSP Page authors, and JSP Tag Library authors.

The chapters are normative.

The chapters are

- JSP Container
- Core API
- Tag Extension API
- Expression Language API

CHAPTER JSP.11

JSP Container

This chapter describes the contracts between a JSP container and a JSP page, including the precompilation protocol and debugging support requirements.

The information in this chapter is independent of the Scripting Language used in the JSP page. Chapter JSP.9 describes information specific to when the language attribute of the page directive has java as its value.).

JSP page implementation classes should use the JspFactory and PageContext classes to take advantage of platform-specific implementations.

JSP.11.1 JSP Page Model

A JSP page is represented at execution time by a JSP page implementation object and is executed by a JSP container. The JSP page implementation object is a servlet. The JSP container delivers requests from a client to a JSP page implementation object and responses from the JSP page implementation object to the client.

The JSP page describes how to create a *response* object from a *request* object for a given protocol, possibly creating and/or using some other objects in the process . A JSP page may also indicate how some events are to be handled. In JSP 2.0 only init and destroy events are allowed events.

JSP.11.1.1 Protocol Seen by the Web Server

The JSP container locates the appropriate instance of the JSP page implementation class and delivers requests to it using the servlet protocol. A JSP container may need to create such a class dynamically from the JSP page source before delivering request and response objects to it.

The Servlet class defines the contract between the JSP container and the JSP page implementation class. When the HTTP protocol is used, the contract is

described by the HttpServlet class. Most JSP pages use the HTTP protocol, but other protocols are allowed by this specification.

The JSP container automatically makes a number of server-side objects available to the JSP page implementation object . See Section JSP.1.8.3.

JSP.11.1.1.1 Protocol Seen by the JSP Page Author

The JSP specification defines the contract between the JSP container and the JSP page author. This contract defines the assumptions an author can make for the actions described in the JSP page.

The main portion of this contract is the _jspService method that is generated automatically by the JSP container from the JSP page. The details of this contract are provided in Chapter JSP.9.

The contract also describes how a JSP author can indicate what actions will be taken when the init and destroy methods of the page implementation occur. In JSP 2.0 this is done by defining methods with the names jspInit and jspDestroy in a declaration scripting element in the JSP page. The jspInit method, if present, will be called to prepare the page before the first request is delivered. Similarly a JSP container can reclaim resources used by a JSP page when a request is not being serviced by the JSP page by invoking its jspDestroy method, if present.

A JSP page author may not (re)define servlet methods through a declaration scripting element.

The JSP specification reserves names for methods and variables starting with jsp, _jsp, jspx, and _jspx, in any combination of upper and lower case.

JSP.11.1.1.2 The HttpJspPage Interface

The enforcement of the contract between the JSP container and the JSP page author is aided by the requirement that the Servlet class corresponding to the JSP page must implement the javax.servlet.jsp.HttpJspPage interface (or the javax.servlet.jsp.JspPage interface if the protocol is not HTTP).

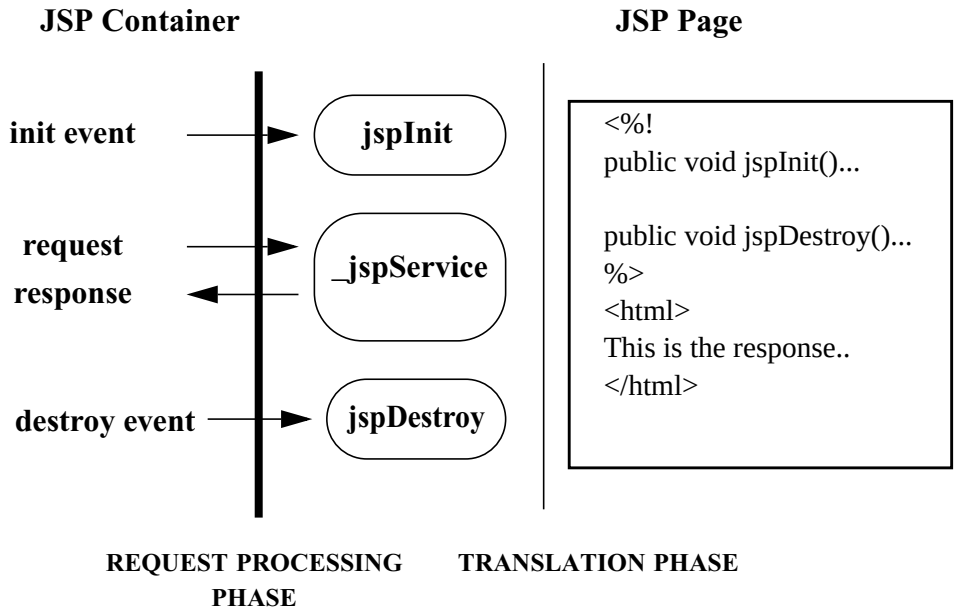


Figure JSP.11-1 Contracts between a JSP Page and a JSP Container.

The involved contracts are shown in Figure JSP.11-1. We now revisit this whole process in more detail.

JSP.11.2 JSP Page Implementation Class

The JSP container creates a JSP page implementation class for each JSP page.

The name of the JSP page implementation class is implementation dependent.

The JSP Page implementation object belongs to an implementation-dependent named package. The package used may vary between one JSP and another, so minimal assumptions should be made.

As of JSP 2.0, it is illegal to refer to any classes from the unnamed (a.k.a. default) package. This may result in a translation error on some containers, specifically those that run in a JDK 1.4 or greater environment. It is unfortunate, but unavoidable, that this will break compatibility with some older JSP applications. However, as of JDK 1.4, importing classes from the unnamed package is not valid (see http://java.sun.com/j2se/1.4/compatibility.html#source for details). Therefore, for forwards compatibility, applications must not rely on the unnamed package. This restriction also applies for all other cases where classes are referenced, such as when specifying the class name for a tag in a TLD.

The JSP container may create the implementation class for a JSP page, or a superclass may be provided by the JSP page author through the use of the extends attribute in the page directive.

The extends mechanism is available for sophisticated users. It should be used with extreme care as it restricts decisions that a JSP container can make. It may restrict efforts to improve performance, for example.

The JSP page implementation class will implement javax.servlet.Servlet and requests are delivered to the class as per the rules in the Servlet 2.4 specification.

A JSP page implementation class may depend on support classes. If the JSP page implementation class is packaged into a WAR, any dependent classes will have to be included so it will be portable across all JSP containers.

A JSP page author writes a JSP page expecting that the client and the server will communicate using a certain protocol. The JSP container must guarantee that requests to and responses from the page use that protocol. Most JSP pages use HTTP, and their implementation classes must implement the HttpJspPage interface, which extends JspPage. If the protocol is not HTTP, then the class will implement an interface that extends JspPage.

JSP.11.2.1 API Contracts

The contract between the JSP container and a Java class implementing a JSP page corresponds to the Servlet interface. Refer to the Servlet 2.4 specification for details.

The responsibility for adhering to this contract rests on the JSP container implementation if the JSP page does not use the extends attribute of the jsp directive. If the extends attribute of the jsp directive is used, the JSP page author must guarantee that the superclass given in the extends attribute supports this contract.

Table JSP.11-1 *How the JSP Container Processes JSP Pages*

Methods the JSP Container Invokes	Comments
void jspInit()	Method is optionally defined in JSP page. Method is invoked when the JSP page is initialized. When method is called all the methods in servlet, including getServletConfig are available

Table JSP.11-1 *How the JSP Container Processes JSP Pages*

Methods the JSP Container Invokes	Comments
void jspDestroy()	Method is optionally defined in JSP page. Method is invoked before destroying the page.
void _jspService(<ServletRequestSubtype>, <ServletResponseSubtype>) throws IOException, ServletException	Method may **not** be defined in JSP page. The JSP container automatically generates this method, based on the contents of the JSP page. Method invoked at each client request.

JSP.11.2.2 Request and Response Parameters

As shown in Table JSP.11-1, the methods in the contract between the JSP container and the JSP page require request and response parameters.

The formal type of the request parameter (which this specification calls <ServletRequestSubtype>) is an interface that extends javax.servlet.ServletRequest. The interface must define a protocol-dependent request contract between the JSP container and the class that implements the JSP page.

Likewise, the formal type of the response parameter (which this specification calls <ServletResponseSubtype>) is an interface that extends javax.servlet.ServletResponse. The interface must define a protocol-dependent response contract between the JSP container and the class that implements the JSP page.

The request and response interfaces together describe a protocol-dependent contract between the JSP container and the class that implements the JSP page. The HTTP contract is defined by the javax.servlet.http.HttpServletRequest and javax.servlet.http.HttpServletResponse interfaces.

The JspPage interface refers to these methods, but cannot describe syntactically the methods involving the Servlet(Request,Response) subtypes. However, interfaces for specific protocols that extend JspPage can, just as HttpJspPage describes them for the HTTP protocol.

JSP containers that conform to this specification (in both JSP page implementation classes and JSP container runtime) must support the request and response interfaces for the HTTP protocol as described in this section.

JSP.11.2.3 Omitting the extends Attribute

If the extends attribute of the page directive (see Section 1.10.1) in a JSP page is not used, the JSP container can generate any class that satisfies the contract described in Table JSP.11-1, when it transforms the JSP page.

In the following code examples, Code Example JSP.11-1 illustrates a generic HTTP superclass named ExampleHttpSuper. Code Example JSP.11-2 shows a subclass named _jsp1344 that extends ExampleHttpSuper and is the class generated from the JSP page. By using separate _jsp1344 and ExampleHttpSuper classes, the JSP page translator does not need to discover whether the JSP page includes a declaration with jspInit or jspDestroy. This significantly simplifies the implementation.

Code Example JSP.11-1 A Generic HTTP Superclass

```
imports javax.servlet.*;
imports javax.servlet.http.*;
imports javax.servlet.jsp.*;

/**
 * An example of a superclass for an HTTP JSP class
 */

abstract class ExampleHttpSuper implements HttpJspPage {
    private ServletConfig config;

final public void init(ServletConfig config) throws ServletException {
    this.config = config;
    jspInit();
}

public void jspInit() {
}

public void jspDestroy() {
}

}

final public ServletConfig getServletConfig() {
    return config;
}
```

```
// This one is not final so it can be overridden by a more precise method
public String getServletInfo() {
      return “A Superclass for an HTTP JSP”; // maybe better?
}

final public void destroy() {
      jspDestroy();
}

/**
* The entry point into service.
*/

final public void service(ServletRequest req, ServletResponse res)
      throws ServletException, IOException {

      // casting exceptions will be raised if an internal error.
      HttpServletRequest request = (HttpServletRequest) req;
      HttpServletResponse response = (HttpServletResponse) res;

      _jspService(request, response);
}

/**
* abstract method to be provided by the JSP processor in the subclass
* Must be defined in subclass.
*/

abstract public void _jspService(HttpServletRequest request,
      HttpServletResponse response) throws ServletException, IOException;

}
```

Code Example JSP.11-2 The Java Class Generated From a JSP Page

```
imports javax.servlet.*;
imports javax.servlet.http.*;
imports javax.servlet.jsp.*;

/**
 * An example of a class generated for a JSP.
 *
 * The name of the class is unpredictable.
 * We are assuming that this is an HTTP JSP page (like almost all are)
 */

class _jsp1344 extends ExampleHttpSuper {

// Next code inserted directly via declarations.
// Any of the following pieces may or not be present
// if they are not defined here the superclass methods
// will be used.

    public void jspInit() {....}
    public void jspDestroy() {....}

    // The next method is generated automatically by the
    // JSP processor.
    // body of JSP page

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

        // initialization of the implicit variables
        // ...

        // next is code from scriptlets, expressions, and static text.

    }

}
```

JSP.11.2.4 Using the extends Attribute

If the JSP page author uses extends, the generated class is identical to the one shown in Code Example JSP.11-2, except that the class name is the one specified in the extends attribute.

The contract on the JSP page implementation class does not change. The JSP container should check (usually through reflection) that the provided superclass:

- Implements HttpJspPage if the protocol is HTTP, or JspPage otherwise.
- All of the methods in the Servlet interface are declared final.

Additionally, it is the responsibility of the JSP page author that the provided superclass satisfies:

- The service method of the servlet API invokes the _jspService method.
- The init(ServletConfig) method stores the configuration, makes it available via getServletConfig, then invokes jspInit.
- The destroy method invokes jspDestroy.

A JSP container may give a fatal translation error if it detects that the provided superclass does not satisfy these requirements, but most JSP containers will not check them.

JSP.11.3 Buffering

The JSP container buffers data (if the jsp directive specifies it using the buffer attribute) as it is sent from the server to the client. Headers are not sent to the client until the first flush method is invoked. Therefore, it is possible to call methods that modify the response header, such as setContentType, sendRedirect, or error methods, up until the flush method is executed and the headers are sent. After that point, these methods become invalid, as per the Servlet specification.

The javax.servlet.jsp.JspWriter class buffers and sends output. The JspWriter class is used in the _jspService method as in the following example:

```
import javax.servlet.jsp.JspWriter;

static JspFactory _jspFactory = JspFactory.getDefaultFactory();

_jspService(<SRequest> request, <SResponse> response) {

    // initialization of implicit variables...
    PageContext pageContext = _jspFactory.createPageContext(
                                this,
                                request,
                                response,
                                false,
                                PageContext.DEFAULT_BUFFER,
                                false
                            );
    JSPWriter out = pageContext.getOut();
    // ....
    // .... the body goes here using "out"
    // ....
    out.flush();
}
```

The complete listing of javax.servlet.jsp.JspWriter can be found in Chapter JSP.12.

With buffering turned on, a redirect method can still be used in a scriptlet in a .jsp file, by invoking response.redirect(someURL) directly.

JSP.11.4 Precompilation

A JSP page that is using the HTTP protocol will receive HTTP requests. JSP 2.0 compliant containers must support a simple *precompilation protocol*, as well as some basic *reserved parameter names*. Note that the precompilation protocol is related but not the same as the notion of compiling a JSP page into a Servlet class (Appendix JSP.A).

JSP.11.4.1 Request Parameter Names

All request parameter names that start with the prefix jsp are reserved by the JSP specification and should not be used by any user or implementation except as indicated by the specification.

All JSPs pages should ignore (not depend on) any parameter that starts with jsp_.

JSP.11.4.2 Precompilation Protocol

A request to a JSP page that has a request parameter with name jsp_precompile is a *precompilation request.* The jsp_precompile parameter may have no value, or may have values true or false. In all cases, the request should not be delivered to the JSP page.

The intention of the precompilation request is that of a suggestion to the JSP container to precompile the JSP page into its JSP page implementation class. The suggestion is conveyed by giving the parameter the value true or no value, but note that the request can be ignored.

For example:

1. ?jsp_precompile
2. ?jsp_precompile=true
3. ?jsp_precompile=false
4. ?foobar=foobaz&jsp_precompile=true
5. ?foobar=foobaz&jsp_precompile=false

   1, 2, and 4 are legal; the request will not be delivered to the page. 3 and 5 are legal; the request will not be delivered to the page.

6. ?jsp_precompile=foo

   This is illegal and will generate an HTTP error; 500 (Server error).

JSP.11.5 Debugging Requirements

With the completion of JSR-045 ("Debugging Support for Other Languages"), the JSP Compiler now has a standard format to convey source map debugging information to tools such as debuggers. See http://jcp.org/jsr/detail/45.jsp for details.

JSP 2.0 containers are strongly recommended, but not required, to be capable of generating source map debugging information for JSP pages and tag files written in either standard or XML syntax. The JSP compiler should be able to produce .class files with a SourceDebugExtension attribute, mapping each line or lines of JSP code to the corresponding generated line or lines of Java code. For both pages and tag files, the stratum that maps to the original source should be

named JSP in the Source Debug Extension (this stratum name is reserved for use by the JSP specification). This stratum should be specified as the default, unless the page or tag file was generated from some other source.

The exact mechanism for causing the JSP compiler to produce source map debugging information is currently implementation-dependent. Full runtime support for JSR-45 (as opposed to only generating the SMAPs that are used at runtime) will typically only be supported when the JSP container is running in a J2SE 1.4 or greater environment.

JSP.11.5.1 Line Number Mapping Guidelines

The following is a set of non-normative guidelines for generating high quality line number mappings. The guidelines are presented to help produce a consistent debugging experience for page authors, across containers. Where possible the JSP container should generate line number mappings as follows:

1. Abreakpoint on a JSP line causes execution to stop before any Java code which amounts to a translation of the JSP line is executed (for one possible exception, see 5). Note that given the LineInfo Composition Algorithm (see JSR-45 specification), it is acceptable for the mappings to include one or more Java lines which are never translated into executable byte code, as long as at least one of them does.
2. It is permitted for two or more lines of JSP to include the same Java lines in their mappings.
3. If a line of JSP has no manifestation in the Java source other than white-space preserving source, it should not be mapped.
   - The following standard syntax JSP entities should not be mapped to generated code. These entities either have no manifestation in the generated Java code (e.g. comments), or are not manifest in such a way that it allows the debugged process to stop (e.g. the page directive import):
     - JSP comments
     - Directives
   - The following XML syntax JSP entities should not be mapped to generated code. These entities frequently have no manifestation in the generated Java code.
     - <jsp:root>
     - <jsp:output>
4. Declarations and scriptlets (standard or XML JSP). Lines in these constructs

should preserve a one-to-one mapping with the corresponding generated code lines. Empty lines and comment lines are not mapped.

5. For scriptlets, scriptlet expressions, EL expressions, standard actions and custom actions in template text, a line containing one or more of these entities should be mapped to Java source lines which include the corresponding Java code.

   If the line starts with template text, the Java code which handles it may be excluded from the mappings if this would cause the debugger to stop before the apparent execution of JSP lines preceding the line in question. For example:

```
100     <p>This is a line with template text.</p>
101     <h1><fmt:message key="company" bundle="${bundle}"/></h1>

200     out.write( "<p>This is a line with template text.</p>\r\n" );
201     out.write( "<h1>" );
202     org.apache.taglibs.standard.tag.el.fmt.MessageTag taghandler =
203         new org.apache.taglibs.standard.tag.el.fmt.MessageTag();
204     taghandler.setPageContext( pageContext );
205     ...
```

   In this example, given that <h1> has its own call to write(), it makes sense to map 101 to 201, 202 etc.

```
200     out.write( "<p>This is a line with template text.</p>\r\n<h1>" );
201     org.apache.taglibs.standard.tag.el.fmt.MessageTag taghandler =
202         new org.apache.taglibs.standard.tag.el.fmt.MessageTag();
203     taghandler.setPageContext( pageContext );
204     ...
```

   In this second example, given that <h1> is output using the same call to write() that was used for line 100, mapping 101 to 202, 203 etc. may result in more intuitive behavior of the debugger.

   For scriptlets that contain more than one line, there should be a one-to-one mapping from JSP to Java lines, and the mapping should start at the first Java code that is not whitespace or comments. Therefore, a line that contains only the open scriptlet delimeter is not mapped.

6. Scriptlet expressions and EL expressions in attribute values. The source line mappings should include any Java source lines that deal with the evaluation of the rtexpr value as well as source that deals with the JSP action.

7. Standard or custom actions.
   - Empty tags and start tags special case: The jsp:params action typically has no manifestation and should not be mapped.
   - Empty tags and start tags: The Java line mappings should include as much of the corresponding Java code as possible, including any separate lines that deal with rtexpr evaluation as described in (6). If it is not possible to include all the Java code in the mappings, the mapped lines should include the first sequential line which deals with either the tag or the attribute evaluation in order to meet (1).
   - Closing tags frequently do not have a manifestation in the Java source, but sometimes do. In case a JSP line contains only a closing tag, the line may be mapped to whitespace preserving Java source if it has no semantic translation. This will avoid a confusing user experience where it is sometimes possible to set a breakpoint on a line consisting of a closing tag and sometimes not.

CHAPTER JSP.12

Core API

This chapter describes the javax.servlet.jsp package. The chapter includes content that is generated automatically from Javadoc embedded into the actual Java classes and interfaces. This allows the creation of a single, authoritative, specification document.

The javax.servlet.jsp package contains a number of classes and interfaces that describe and define the contracts between a JSP page implementation class and the runtime environment provided for an instance of such a class by a conforming JSP container.

JSP.12.1 JSP Page Implementation Object Contract

This section describes the basic contract between a JSP Page implementation object and its container. The main contract is defined by the classes JspPage and HttpJspPage. The JspFactory class describes the mechanism to portably instantiate all needed runtime objects, and JspEngineInfo provides basic information on the current JSP container.

None of the classes described here are intended to be used by JSP page authors; an example of how these classes may be used is included elsewhere in this chapter.

JSP.12.1.1 JspPage

Syntax

public interface JspPage extends javax.servlet.Servlet

All Known Subinterfaces: HttpJspPage

All Superinterfaces: javax.servlet.Servlet

Description

The JspPage interface describes the generic interaction that a JSP Page Implementation class must satisfy; pages that use the HTTP protocol are described by the HttpJspPage interface.

Two plus One Methods

The interface defines a protocol with 3 methods; only two of them: jspInit() and jspDestroy() are part of this interface as the signature of the third method: _jspService() depends on the specific protocol used and cannot be expressed in a generic way in Java.

A class implementing this interface is responsible for invoking the above methods at the appropriate time based on the corresponding Servlet-based method invocations.

The jspInit() and jspDestroy() methods can be defined by a JSP author, but the _jspService() method is defined automatically by the JSP processor based on the contents of the JSP page.

_jspService()

The _jspService()method corresponds to the body of the JSP page. This method is defined automatically by the JSP container and should never be defined by the JSP page author.

If a superclass is specified using the extends attribute, that superclass may choose to perform some actions in its service() method before or after calling the _jspService() method. See using the extends attribute in the JSP_Engine chapter of the JSP specification.

The specific signature depends on the protocol supported by the JSP page.

```
public void _jspService(ServletRequestSubtype request,
                        ServletResponseSubtype response)
    throws ServletException, IOException;
```

JSP.12.1.1.1 Methods

public void **jspDestroy**()

The jspDestroy() method is invoked when the JSP page is about to be destroyed. A JSP page can override this method by including a definition for it in a declaration element. A JSP page should redefine the destroy() method from Servlet.

public void **jspInit**()

The jspInit() method is invoked when the JSP page is initialized. It is the responsibility of the JSP implementation (and of the class mentioned by the extends attribute, if present) that at this point invocations to the getServlet-Config() method will return the desired value. A JSP page can override this method by including a definition for it in a declaration element. A JSP page should redefine the init() method from Servlet.

JSP.12.1.2 HttpJspPage

Syntax

public interface HttpJspPage extends JspPage

All Superinterfaces: JspPage, javax.servlet.Servlet

Description

The HttpJspPage interface describes the interaction that a JSP Page Implementation Class must satisfy when using the HTTP protocol.

The behaviour is identical to that of the JspPage, except for the signature of the _jspService method, which is now expressible in the Java type system and included explicitly in the interface.

See Also: JspPage

JSP.12.1.2.1 Methods

public void _**jspService**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response)

The _jspService()method corresponds to the body of the JSP page. This method is defined automatically by the JSP container and should never be defined by the JSP page author.

If a superclass is specified using the extends attribute, that superclass may choose to perform some actions in its service() method before or after calling the _jspService() method. See using the extends attribute in the JSP_Engine chapter of the JSP specification.

Parameters:
request - Provides client request information to the JSP.

response - Assists the JSP in sending a response to the client.

Throws:
ServletException - Thrown if an error occurred during the processing of the JSP and that the container should take appropriate action to clean up the request.

IOException - Thrown if an error occurred while writing the response for this page.

JSP.12.1.3 JspFactory

Syntax

public abstract class JspFactory

Description

The JspFactory is an abstract class that defines a number of factory methods available to a JSP page at runtime for the purposes of creating instances of various interfaces and classes used to support the JSP implementation.

A conformant JSP Engine implementation will, during it's initialization instantiate an implementation dependent subclass of this class, and make it globally available for use by JSP implementation classes by registering the instance created with this class via the static setDefaultFactory() method.

The PageContext and the JspEngineInfo classes are the only implementation-dependent classes that can be created from the factory.

JspFactory objects should not be used by JSP page authors.

JSP.12.1.3.1 Constructors

public **JspFactory**()

Sole constructor. (For invocation by subclass constructors, typically implicit.)

JSP.12.1.3.2 Methods

public static synchronized JspFactory **getDefaultFactory**()

Returns the default factory for this implementation.

Returns: the default factory for this implementation

public abstract JspEngineInfo **getEngineInfo**()

called to get implementation-specific information on the current JSP engine.

Returns: a JspEngineInfo object describing the current JSP engine

public abstract PageContext **getPageContext**(javax.servlet.Servlet servlet, javax.servlet.ServletRequest request, javax.servlet.ServletResponse response, java.lang.String errorPageURL, boolean needsSession, int buffer, boolean autoflush)

obtains an instance of an implementation dependent javax.servlet.jsp.PageContext abstract class for the calling Servlet and currently pending request and response.

This method is typically called early in the processing of the _jspService() method of a JSP implementation class in order to obtain a PageContext object for the request being processed.

Invoking this method shall result in the PageContext.initialize() method being invoked. The PageContext returned is properly initialized.

All PageContext objects obtained via this method shall be released by invoking releasePageContext().

Parameters:
servlet - the requesting servlet

request - the current request pending on the servlet

response - the current response pending on the servlet

errorPageURL - the URL of the error page for the requesting JSP, or null

needsSession - true if the JSP participates in a session

buffer - size of buffer in bytes, PageContext.NO_BUFFER if no buffer, PageContext.DEFAULT_BUFFER if implementation default.

autoflush - should the buffer autoflush to the output stream on buffer overflow, or throw an IOException?

Returns: the page context

See Also: PageContext

public abstract void **releasePageContext**(PageContext pc)

called to release a previously allocated PageContext object. Results in PageContext.release() being invoked. This method should be invoked prior to returning from the _jspService() method of a JSP implementation class.

Parameters:
pc - A PageContext previously obtained by getPageContext()

public static synchronized void **setDefaultFactory**(JspFactory deflt)

set the default factory for this implementation. It is illegal for any principal other than the JSP Engine runtime to call this method.

Parameters:

deflt - The default factory implementation

JSP.12.1.4 JspEngineInfo

Syntax

public abstract class JspEngineInfo

Description

The JspEngineInfo is an abstract class that provides information on the current JSP engine.

JSP.12.1.4.1 Constructors

public **JspEngineInfo**()

Sole constructor. (For invocation by subclass constructors, typically implicit.)

JSP.12.1.4.2 Methods

public abstract java.lang.String **getSpecificationVersion**()

Return the version number of the JSP specification that is supported by this JSP engine.

Specification version numbers that consists of positive decimal integers separated by periods ".", for example, "2.0" or "1.2.3.4.5.6.7". This allows an extensible number to be used to represent major, minor, micro, etc versions. The version number must begin with a number.

Returns: the specification version, null is returned if it is not known

JSP.12.2 Implicit Objects

The PageContext object and the JspWriter are available by default as implicit objects.

JSP.12.2.1 JspContext

Syntax

public abstract class JspContext

Direct Known Subclasses: PageContext

Description

JspContext serves as the base class for the PageContext class and abstracts all information that is not specific to servlets. This allows for Simple Tag Extensions to be used outside of the context of a request/response Servlet.

The JspContext provides a number of facilities to the page/component author and page implementor, including:

- a single API to manage the various scoped namespaces
- a mechanism to obtain the JspWriter for output
- a mechanism to expose page directive attributes to the scripting environment

Methods Intended for Container Generated Code

The following methods enable the **management of nested** JspWriter streams to implement Tag Extensions: pushBody() and popBody()

Methods Intended for JSP authors

Some methods provide **uniform access** to the diverse objects representing scopes. The implementation must use the underlying machinery corresponding to that scope, so information can be passed back and forth between the underlying environment (e.g. Servlets) and JSP pages. The methods are: setAttribute(), getAttribute(), findAttribute(), removeAttribute(), getAttributesScope() and getAttributeNamesInScope().

The following methods provide **convenient access** to implicit objects: getOut()

The following methods provide **programmatic access** to the Expression Language evaluator: getExpressionEvaluator(), getVariableResolver()

Since: 2.0

JSP.12.2.1.1 Constructors

public **JspContext**()

Sole constructor. (For invocation by subclass constructors, typically implicit.)

JSP.12.2.1.2 Methods

public abstract java.lang.Object **findAttribute**(java.lang.String name)

Searches for the named attribute in page, request, session (if valid), and application scope(s) in order and returns the value associated or null.

Parameters:

name - the name of the attribute to search for

Returns: the value associated or null

Throws:
NullPointerException - if the name is null

public abstract java.lang.Object **getAttribute**(java.lang.String name)

Returns the object associated with the name in the page scope or null if not found.

Parameters:
name - the name of the attribute to get

Returns: the object associated with the name in the page scope or null if not found.

Throws:
NullPointerException - if the name is null

public abstract java.lang.Object **getAttribute**(java.lang.String name, int scope)

Return the object associated with the name in the specified scope or null if not found.

Parameters:
name - the name of the attribute to set

scope - the scope with which to associate the name/object

Returns: the object associated with the name in the specified scope or null if not found.

Throws:
NullPointerException - if the name is null

IllegalArgumentException - if the scope is invalid

IllegalStateException - if the scope is PageContext.SESSION_SCOPE but the page that was requested does not participate in a session or the session has been invalidated.

public abstract java.util.Enumeration **getAttributeNamesInScope**(int scope)

Enumerate all the attributes in a given scope.

Parameters:
scope - the scope to enumerate all the attributes for

Returns: an enumeration of names (java.lang.String) of all the attributes the specified scope

Throws:
IllegalArgumentException - if the scope is invalid

IllegalStateException - if the scope is PageContext.SESSION_SCOPE but the page that was requested does not participate in a session or the session has been invalidated.

public abstract int **getAttributesScope**(java.lang.String name)

Get the scope where a given attribute is defined.

Parameters:
name - the name of the attribute to return the scope for

Returns: the scope of the object associated with the name specified or 0

Throws:
NullPointerException - if the name is null

public abstract ExpressionEvaluator **getExpressionEvaluator**()

Provides programmatic access to the ExpressionEvaluator. The JSP Container must return a valid instance of an ExpressionEvaluator that can parse EL expressions.

Returns: A valid instance of an ExpressionEvaluator.

Since: 2.0

public abstract JspWriter **getOut**()

The current value of the out object (a JspWriter).

Returns: the current JspWriter stream being used for client response

public abstract VariableResolver **getVariableResolver**()

Returns an instance of a VariableResolver that provides access to the implicit objects specified in the JSP specification using this JspContext as the context object.

Returns: A valid instance of a VariableResolver.

Since: 2.0

public JspWriter **popBody**()

Return the previous JspWriter “out” saved by the matching pushBody(), and update the value of the “out” attribute in the page scope attribute namespace of the JspContext.

Returns: the saved JspWriter.

public JspWriter **pushBody**(java.io.Writer writer)

Return a new JspWriter object that sends output to the provided Writer. Saves the current “out” JspWriter, and updates the value of the “out” attribute in the page scope attribute namespace of the JspContext.

The returned JspWriter must implement all methods and behave as though it were unbuffered. More specifically:

- clear() must throw an IOException
- clearBuffer() does nothing
- getBufferSize() always returns 0
- getRemaining() always returns 0

Parameters:
writer - The Writer for the returned JspWriter to send output to.

Returns: a new JspWriter that writes to the given Writer.

Since: 2.0

public abstract void **removeAttribute**(java.lang.String name)

Remove the object reference associated with the given name from all scopes. Does nothing if there is no such object.

Parameters:
name - The name of the object to remove.

Throws:
NullPointerException - if the name is null

public abstract void **removeAttribute**(java.lang.String name, int scope)

Remove the object reference associated with the specified name in the given scope. Does nothing if there is no such object.

Parameters:
name - The name of the object to remove.

scope - The scope where to look.

Throws:
IllegalArgumentException - if the scope is invalid

IllegalStateException - if the scope is PageContext.SESSION_SCOPE but the page that was requested does not participate in a session or the session has been invalidated.

NullPointerException - if the name is null

public abstract void **setAttribute**(java.lang.String name, java.lang.Object value)

Register the name and value specified with page scope semantics. If the value passed in is null, this has the same effect as calling removeAttribute(name, PageContext.PAGE_SCOPE).

Parameters:
name - the name of the attribute to set

value - the value to associate with the name, or null if the attribute is to be removed from the page scope.

Throws:
NullPointerException - if the name is null

public abstract void **setAttribute**(java.lang.String name, java.lang.Object value, int scope)

Register the name and value specified with appropriate scope semantics. If the value passed in is null, this has the same effect as calling removeAttribute(name, scope).

Parameters:
name - the name of the attribute to set

value - the object to associate with the name, or null if the attribute is to be removed from the specified scope.

scope - the scope with which to associate the name/object

Throws:
NullPointerException - if the name is null

IllegalArgumentException - if the scope is invalid

IllegalStateException - if the scope is PageContext.SESSION_SCOPE but the page that was requested does not participate in a session or the session has been invalidated.

JSP.12.2.2 PageContext

Syntax

public abstract class PageContext extends JspContext

Description

PageContext extends JspContext to provide useful context information for when JSP technology is used in a Servlet environment.

A PageContext instance provides access to all the namespaces associated with a JSP page, provides access to several page attributes, as well as a layer above the implementation details. Implicit objects are added to the pageContext automatically.

The PageContext class is an abstract class, designed to be extended to provide implementation dependent implementations thereof, by conformant JSP engine runtime environments. A PageContext instance is obtained by a JSP implementa-

tion class by calling the JspFactory.getPageContext() method, and is released by calling JspFactory.releasePageContext().

An example of how PageContext, JspFactory, and other classes can be used within a JSP Page Implementation object is given elsewhere.

The PageContext provides a number of facilities to the page/component author and page implementor, including:

- a single API to manage the various scoped namespaces
- a number of convenience API's to access various public objects
- a mechanism to obtain the JspWriter for output
- a mechanism to manage session usage by the page
- a mechanism to expose page directive attributes to the scripting environment
- mechanisms to forward or include the current request to other active components in the application
- a mechanism to handle errorpage exception processing

Methods Intended for Container Generated Code

Some methods are intended to be used by the code generated by the container, not by code written by JSP page authors, or JSP tag library authors.

The methods supporting **lifecycle** are initialize() and release()

The following methods enable the **management of nested** JspWriter streams to implement Tag Extensions: pushBody()

Methods Intended for JSP authors

The following methods provide **convenient access** to implicit objects: getException(), getPage() getRequest(), getResponse(), getSession(), getServletConfig() and getServletContext().

The following methods provide support for **forwarding, inclusion and error handling**: forward(), include(), and handlePageException().

JSP.12.2.2.1 Fields

public static final java.lang.String **APPLICATION**

Name used to store ServletContext in PageContext name table.

public static final int **APPLICATION_SCOPE**

Application scope: named reference remains available in the ServletContext until it is reclaimed.

public static final java.lang.String **CONFIG**

Name used to store ServletConfig in PageContext name table.

public static final java.lang.String **EXCEPTION**

Name used to store uncaught exception in ServletRequest attribute list and PageContext name table.

public static final java.lang.String **OUT**

Name used to store current JspWriter in PageContext name table.

public static final java.lang.String **PAGE**

Name used to store the Servlet in this PageContext's nametables.

public static final int **PAGE_SCOPE**

Page scope: (this is the default) the named reference remains available in this PageContext until the return from the current Servlet.service() invocation.

public static final java.lang.String **PAGECONTEXT**

Name used to store this PageContext in it's own name table.

public static final java.lang.String **REQUEST**

Name used to store ServletRequest in PageContext name table.

public static final int **REQUEST_SCOPE**

Request scope: the named reference remains available from the Servlet-Request associated with the Servlet until the current request is completed.

public static final java.lang.String **RESPONSE**

Name used to store ServletResponse in PageContext name table.

public static final java.lang.String **SESSION**

Name used to store HttpSession in PageContext name table.

public static final int **SESSION_SCOPE**

Session scope (only valid if this page participates in a session): the named reference remains available from the HttpSession (if any) associated with the Servlet until the HttpSession is invalidated.

JSP.12.2.2.2 Constructors

public **PageContext**()

Sole constructor. (For invocation by subclass constructors, typically implicit.)

JSP.12.2.2.3 Methods

public abstract void **forward**(java.lang.String relativeUrlPath)

This method is used to re-direct, or "forward" the current ServletRequest and ServletResponse to another active component in the application.

If the *relativeUrlPath* begins with a "/" then the URL specified is calculated relative to the DOCROOT of the ServletContext for this JSP. If the path does not begin with a "/" then the URL specified is calculated relative to the URL of the request that was mapped to the calling JSP.

It is only valid to call this method from a Thread executing within a _jspService(...) method of a JSP.

Once this method has been called successfully, it is illegal for the calling Thread to attempt to modify the ServletResponse object. Any such attempt to do so, shall result in undefined behavior. Typically, callers immediately return from _jspService(...) after calling this method.

Parameters:
relativeUrlPath - specifies the relative URL path to the target resource as described above

Throws:
IllegalStateException - if ServletResponse is not in a state where a forward can be performed

ServletException - if the page that was forwarded to throws a ServletException

IOException - if an I/O error occurred while forwarding

public ErrorData **getErrorData**()

Provides convenient access to error information.

Returns: an ErrorData instance containing information about the error, as obtained from the request attributes, as per the Servlet specification. If this is not an error page (that is, if the isErrorPage attribute of the page directive is not set to "true"), the information is meaningless.

Since: 2.0

public abstract java.lang.Exception **getException**()

The current value of the exception object (an Exception).

Returns: any exception passed to this as an errorpage

public abstract java.lang.Object **getPage**()

The current value of the page object (In a Servlet environment, this is an instance of javax.servlet.Servlet).

Returns: the Page implementation class instance associated with this PageContext

public abstract javax.servlet.ServletRequest **getRequest**()

The current value of the request object (a ServletRequest).

Returns: The ServletRequest for this PageContext

public abstract javax.servlet.ServletResponse **getResponse**()

The current value of the response object (a ServletResponse).

Returns: the ServletResponse for this PageContext

public abstract javax.servlet.ServletConfig **getServletConfig**()

The ServletConfig instance.

Returns: the ServletConfig for this PageContext

public abstract javax.servlet.ServletContext **getServletContext**()

The ServletContext instance.

Returns: the ServletContext for this PageContext

public abstract javax.servlet.http.HttpSession **getSession**()

The current value of the session object (an HttpSession).

Returns: the HttpSession for this PageContext or null

public abstract void **handlePageException**(java.lang.Exception e)

This method is intended to process an unhandled 'page' level exception by forwarding the exception to the specified error page for this JSP. If forwarding is not possible (for example because the response has already been committed), an implementation dependent mechanism should be used to invoke the error page (e.g. "including" the error page instead).

If no error page is defined in the page, the exception should be rethrown so that the standard servlet error handling takes over.

A JSP implementation class shall typically clean up any local state prior to invoking this and will return immediately thereafter. It is illegal to generate any output to the client, or to modify any ServletResponse state after invoking this call.

This method is kept for backwards compatiblity reasons. Newly generated code should use PageContext.handlePageException(Throwable).

Parameters:
e - the exception to be handled

Throws:
ServletException - if an error occurs while invoking the error page

IOException - if an I/O error occurred while invoking the error page

NullPointerException - if the exception is null

See Also: public abstract void handlePageException(java.lang.Throwable t)

public abstract void **handlePageException**(java.lang.Throwable t)

This method is intended to process an unhandled 'page' level exception by forwarding the exception to the specified error page for this JSP. If forwarding is not possible (for example because the response has already been committed), an implementation dependent mechanism should be used to invoke the error page (e.g. "including" the error page instead).

If no error page is defined in the page, the exception should be rethrown so that the standard servlet error handling takes over.

This method is intended to process an unhandled "page" level exception by redirecting the exception to either the specified error page for this JSP, or if none was specified, to perform some implementation dependent action.

A JSP implementation class shall typically clean up any local state prior to invoking this and will return immediately thereafter. It is illegal to generate any output to the client, or to modify any ServletResponse state after invoking this call.

Parameters:
t - the throwable to be handled

Throws:
ServletException - if an error occurs while invoking the error page

IOException - if an I/O error occurred while invoking the error page

NullPointerException - if the exception is null

See Also: public abstract void handlePageException(java.lang.Exception e)

public abstract void **include**(java.lang.String relativeUrlPath)

Causes the resource specified to be processed as part of the current ServletRequest and ServletResponse being processed by the calling Thread. The output of the target resources processing of the request is written directly to the ServletResponse output stream.

The current JspWriter "out" for this JSP is flushed as a side-effect of this call, prior to processing the include.

If the *relativeUrlPath* begins with a "/" then the URL specified is calculated relative to the DOCROOT of the ServletContext for this JSP. If the path does not begin with a "/" then the URL specified is calculated relative to the URL of the request that was mapped to the calling JSP.

It is only valid to call this method from a Thread executing within a _jspService(...) method of a JSP.

Parameters:
relativeUrlPath - specifies the relative URL path to the target resource to be included

Throws:
ServletException - if the page that was forwarded to throws a ServletException

IOException - if an I/O error occurred while forwarding

public abstract void **include**(java.lang.String relativeUrlPath, boolean flush)

Causes the resource specified to be processed as part of the current ServletRequest and ServletResponse being processed by the calling Thread. The output of the target resources processing of the request is written directly to the current JspWriter returned by a call to getOut().

If flush is true, The current JspWriter "out" for this JSP is flushed as a side-effect of this call, prior to processing the include. Otherwise, the JspWriter "out" is not flushed.

If the *relativeUrlPath* begins with a "/" then the URL specified is calculated relative to the DOCROOT of the ServletContext for this JSP. If the path does not begin with a "/" then the URL specified is calculated relative to the URL of the request that was mapped to the calling JSP.

It is only valid to call this method from a Thread executing within a _jspService(...) method of a JSP.

Parameters:
relativeUrlPath - specifies the relative URL path to the target resource to be included

flush - True if the JspWriter is to be flushed before the include, or false if not.

Throws:
ServletException - if the page that was forwarded to throws a ServletException

IOException - if an I/O error occurred while forwarding

Since: 2.0

public abstract void **initialize**(javax.servlet.Servlet servlet,
javax.servlet.ServletRequest request,
javax.servlet.ServletResponse response, java.lang.String errorPageURL,
boolean needsSession, int bufferSize, boolean autoFlush)

The initialize method is called to initialize an uninitialized PageContext so that it may be used by a JSP Implementation class to service an incoming request and response within it's _jspService() method.

This method is typically called from JspFactory.getPageContext() in order to initialize state.

This method is required to create an initial JspWriter, and associate the "out" name in page scope with this newly created object.

This method should not be used by page or tag library authors.

Parameters:
servlet - The Servlet that is associated with this PageContext

request - The currently pending request for this Servlet

response - The currently pending response for this Servlet

errorPageURL - The value of the errorpage attribute from the page directive or null

needsSession - The value of the session attribute from the page directive

bufferSize - The value of the buffer attribute from the page directive

autoFlush - The value of the autoflush attribute from the page directive

Throws:
IOException - during creation of JspWriter

IllegalStateException - if out not correctly initialized

IllegalArgumentException - If one of the given parameters is invalid

public BodyContent **pushBody**()

Return a new BodyContent object, save the current "out" JspWriter, and update the value of the "out" attribute in the page scope attribute namespace of the PageContext.

Returns: the new BodyContent

public abstract void **release**()

This method shall "reset" the internal state of a PageContext, releasing all internal references, and preparing the PageContext for potential reuse by a later invocation of initialize(). This method is typically called from JspFactory.releasePageContext().

Subclasses shall envelope this method.

This method should not be used by page or tag library authors.

JSP.12.2.3 JspWriter

Syntax

public abstract class JspWriter extends java.io.Writer

Direct Known Subclasses: BodyContent

Description

The actions and template data in a JSP page is written using the JspWriter object that is referenced by the implicit variable out which is initialized automatically using methods in the PageContext object.

This abstract class emulates some of the functionality found in the java.io.BufferedWriter and java.io.PrintWriter classes, however it differs in that it throws java.io.IOException from the print methods while PrintWriter does not.

Buffering

The initial JspWriter object is associated with the PrintWriter object of the ServletResponse in a way that depends on whether the page is or is not buffered. If the page is not buffered, output written to this JspWriter object will be written through to the PrintWriter directly, which will be created if necessary by invoking the getWriter() method on the response object. But if the page is buffered, the PrintWriter object will not be created until the buffer is flushed and operations like setContentType() are legal. Since this flexibility simplifies programming substantially, buffering is the default for JSP pages.

Buffering raises the issue of what to do when the buffer is exceeded. Two approaches can be taken:

- Exceeding the buffer is not a fatal error; when the buffer is exceeded, just flush the output.
- Exceeding the buffer is a fatal error; when the buffer is exceeded, raise an exception.

Both approaches are valid, and thus both are supported in the JSP technology. The behavior of a page is controlled by the autoFlush attribute, which defaults to true. In general, JSP pages that need to be sure that correct and complete data has been sent to their client may want to set autoFlush to false, with a typical case being that where the client is an application itself. On the other hand, JSP pages that send data that is meaningful even when partially constructed may want to set autoFlush to true; such as when the data is sent for immediate display through a browser. Each application will need to consider their specific needs.

An alternative considered was to make the buffer size unbounded; but, this had the disadvantage that runaway computations would consume an unbounded amount of resources.

The “out” implicit variable of a JSP implementation class is of this type. If the page directive selects autoflush=“true” then all the I/O operations on this class shall automatically flush the contents of the buffer if an overflow condition would result if the current operation were performed without a flush. If autoflush=“false” then all the I/O operations on this class shall throw an IOException if performing the current operation would result in a buffer overflow condition.

See Also: java.io.Writer, java.io.BufferedWriter, java.io.PrintWriter

JSP.12.2.3.1 Fields

protected boolean **autoFlush**

Whether the JspWriter is autoflushing.

protected int **bufferSize**

The size of the buffer used by the JspWriter.

public static final int **DEFAULT_BUFFER**

Constant indicating that the Writer is buffered and is using the implementation default buffer size.

public static final int **NO_BUFFER**

Constant indicating that the Writer is not buffering output.

public static final int **UNBOUNDED_BUFFER**

Constant indicating that the Writer is buffered and is unbounded; this is used in BodyContent.

JSP.12.2.3.2 Constructors

protected **JspWriter**(int bufferSize, boolean autoFlush)

Protected constructor.

Parameters:
bufferSize - the size of the buffer to be used by the JspWriter

autoFlush - whether the JspWriter should be autoflushing

JSP.12.2.3.3 Methods

public abstract void **clear**()

Clear the contents of the buffer. If the buffer has been already been flushed then the clear operation shall throw an IOException to signal the fact that some data has already been irrevocably written to the client response stream.

Throws:
IOException - If an I/O error occurs

public abstract void **clearBuffer**()

Clears the current contents of the buffer. Unlike clear(), this method will not throw an IOException if the buffer has already been flushed. It merely clears the current content of the buffer and returns.

Throws:
IOException - If an I/O error occurs

public abstract void **close**()

Close the stream, flushing it first.

This method needs not be invoked explicitly for the initial JspWriter as the code generated by the JSP container will automatically include a call to close().

Closing a previously-closed stream, unlike flush(), has no effect.

Overrides: java.io.Writer.close() in class java.io.Writer

Throws:
IOException - If an I/O error occurs

public abstract void **flush**()

Flush the stream. If the stream has saved any characters from the various write() methods in a buffer, write them immediately to their intended destination. Then, if that destination is another character or byte stream, flush it. Thus one flush() invocation will flush all the buffers in a chain of Writers and OutputStreams.

The method may be invoked indirectly if the buffer size is exceeded.

Once a stream has been closed, further write() or flush() invocations will cause an IOException to be thrown.

Overrides: java.io.Writer.flush() in class java.io.Writer

Throws:
IOException - If an I/O error occurs

public int **getBufferSize**()

This method returns the size of the buffer used by the JspWriter.

Returns: the size of the buffer in bytes, or 0 is unbuffered.

public abstract int **getRemaining**()

This method returns the number of unused bytes in the buffer.

Returns: the number of bytes unused in the buffer

public boolean **isAutoFlush**()

This method indicates whether the JspWriter is autoFlushing.

Returns: if this JspWriter is auto flushing or throwing IOExceptions on buffer overflow conditions

public abstract void **newLine**()

Write a line separator. The line separator string is defined by the system property line.separator, and is not necessarily a single newline ('\n') character.

Throws:
IOException - If an I/O error occurs

public abstract void **print**(boolean b)

Print a boolean value. The string produced by java.lang.String.valueOf(boolean) is written to the JspWriter's buffer or, if no buffer is used, directly to the underlying writer.

Parameters:
b - The boolean to be printed

Throws:
java.io.IOException - If an error occured while writing

public abstract void **print**(char c)

Print a character. The character is written to the JspWriter's buffer or, if no buffer is used, directly to the underlying writer.

Parameters:
c - The char to be printed

Throws:
java.io.IOException - If an error occured while writing

public abstract void **print**(char[] s)

Print an array of characters. The characters are written to the JspWriter's buffer or, if no buffer is used, directly to the underlying writer.

Parameters:
s - The array of chars to be printed

Throws:
NullPointerException - If s is null

java.io.IOException - If an error occured while writing

public abstract void **print**(double d)

Print a double-precision floating-point number. The string produced by java.lang.String.valueOf(double) is written to the JspWriter's buffer or, if no buffer is used, directly to the underlying writer.

Parameters:
d - The double to be printed

Throws:
java.io.IOException - If an error occured while writing

See Also: java.lang.Double

public abstract void **print**(float f)

Print a floating-point number. The string produced by java.lang.String.valueOf(float) is written to the JspWriter's buffer or, if no buffer is used, directly to the underlying writer.

Parameters:
f - The float to be printed

Throws:
java.io.IOException - If an error occured while writing

See Also: java.lang.Float

public abstract void **print**(int i)

Print an integer. The string produced by java.lang.String.valueOf(int) is written to the JspWriter's buffer or, if no buffer is used, directly to the underlying writer.

Parameters:
i - The int to be printed

Throws:
java.io.IOException - If an error occured while writing

See Also: java.lang.Integer

public abstract void **print**(long l)

Print a long integer. The string produced by java.lang.String.valueOf(long) is written to the JspWriter's buffer or, if no buffer is used, directly to the underlying writer.

Parameters:
l - The long to be printed

Throws:
java.io.IOException - If an error occured while writing

See Also: java.lang.Long

public abstract void **print**(java.lang.Object obj)

Print an object. The string produced by the java.lang.String.valueOf(Object) method is written to the JspWriter's buffer or, if no buffer is used, directly to the underlying writer.

Parameters:
obj - The Object to be printed

Throws:
java.io.IOException - If an error occured while writing

See Also: java.lang.Object.toString()

public abstract void **print**(java.lang.String s)

Print a string. If the argument is null then the string "null" is printed. Otherwise, the string's characters are written to the JspWriter's buffer or, if no buffer is used, directly to the underlying writer.

Parameters:
s - The String to be printed

Throws:
java.io.IOException - If an error occured while writing

public abstract void **println**()

Terminate the current line by writing the line separator string. The line separator string is defined by the system property line.separator, and is not necessarily a single newline character ('\n').

Throws:
java.io.IOException - If an error occured while writing

public abstract void **println**(boolean x)

Print a boolean value and then terminate the line. This method behaves as though it invokes public abstract void print(boolean b) and then public abstract void println() .

Parameters:
x - the boolean to write

Throws:
java.io.IOException - If an error occured while writing

public abstract void **println**(char x)

Print a character and then terminate the line. This method behaves as though it invokes public abstract void print(char c) and then public abstract void println() .

Parameters:

x - the char to write

Throws:
java.io.IOException - If an error occured while writing

public abstract void **println**(char[] x)

Print an array of characters and then terminate the line. This method behaves as though it invokes print(char[]) and then println().

Parameters:
x - the char[] to write

Throws:
java.io.IOException - If an error occured while writing

public abstract void **println**(double x)

Print a double-precision floating-point number and then terminate the line. This method behaves as though it invokes public abstract void print(double d) and then public abstract void println() .

Parameters:
x - the double to write

Throws:
java.io.IOException - If an error occured while writing

public abstract void **println**(float x)

Print a floating-point number and then terminate the line. This method behaves as though it invokes public abstract void print(float f) and then public abstract void println() .

Parameters:
x - the float to write

Throws:
java.io.IOException - If an error occured while writing

public abstract void **println**(int x)

Print an integer and then terminate the line. This method behaves as though it invokes public abstract void print(int i) and then public abstract void println() .

Parameters:
x - the int to write

Throws:
java.io.IOException - If an error occured while writing

public abstract void **println**(long x)

Print a long integer and then terminate the line. This method behaves as though it invokes public abstract void print(long l) and then public abstract void println() .

Parameters:
x - the long to write

Throws:
java.io.IOException - If an error occured while writing

public abstract void **println**(java.lang.Object x)

Print an Object and then terminate the line. This method behaves as though it invokes public abstract void print(java.lang.Object obj) and then public abstract void println() .

Parameters:
x - the Object to write

Throws:
java.io.IOException - If an error occured while writing

public abstract void **println**(java.lang.String x)

Print a String and then terminate the line. This method behaves as though it invokes public abstract void print(java.lang.String s) and then public abstract void println() .

Parameters:
x - the String to write

Throws:
java.io.IOException - If an error occured while writing

JSP.12.2.4 ErrorData

Syntax

public final class ErrorData

Description

Contains information about an error, for error pages. The information contained in this instance is meaningless if not used in the context of an error page. To indicate a JSP is an error page, the page author must set the isErrorPage attribute of the page directive to “true”.

Since: 2.0

See Also: public ErrorData getErrorData()

JSP.12.2.4.1 Constructors

public **ErrorData**(java.lang.Throwable throwable, int statusCode, java.lang.String uri, java.lang.String servletName)

Creates a new ErrorData object.

Parameters:
throwable - The Throwable that is the cause of the error

statusCode - The status code of the error

uri - The request URI

servletName - The name of the servlet invoked

JSP.12.2.4.2 Methods

public java.lang.String **getRequestURI**()

Returns the request URI.

Returns: The request URI

public java.lang.String **getServletName**()

Returns the name of the servlet invoked.

Returns: The name of the servlet invoked

public int **getStatusCode**()

Returns the status code of the error.

Returns: The status code of the error

public java.lang.Throwable **getThrowable**()

Returns the Throwable that caused the error.

Returns: The Throwable that caused the error

JSP.12.3 An Implementation Example

An instance of an implementation dependent subclass of this abstract base class can be created by a JSP implementation class at the beginning of it's _jspService() method via an implementation default JspFactory .

Here is one example of how to use these classes

```
 public class foo implements Servlet {
 // ...
public void _jspService(HttpServletRequest request,
HttpServletResponse response)
      throws IOException, ServletException {
   JspFactory  factory     = JspFactory.getDefaultFactory();
   PageContext pageContext = factory.getPageContext(
this,
request,
response,
null,  // errorPageURL
false, // needsSession
JspWriter.DEFAULT_BUFFER,
true   // autoFlush
       );
   // initialize implicit variables for scripting env ...
   HttpSession session = pageContext.getSession();
   JspWriter   out     = pageContext.getOut();
   Object      page    = this;
   try {
      // body of translated JSP here ...
   } catch (Exception e) {
      out.clear();
      pageContext.handlePageException(e);
   } finally {
      out.close();
  factory.releasePageContext(pageContext);
   }
}
```

JSP.12.4 Exceptions

The JspException class is the base class for all JSP exceptions. The JspTagException and SkipPageException exceptions are used by the tag extension mechanism.

JSP.12.4.1 JspException

Syntax

public class JspException extends java.lang.Exception

Direct Known Subclasses: JspTagException, SkipPageException

All Implemented Interfaces: java.io.Serializable

Description

A generic exception known to the JSP engine; uncaught JspExceptions will result in an invocation of the errorpage machinery.

JSP.12.4.1.1 Constructors

public **JspException**()

Construct a JspException.

public **JspException**(java.lang.String msg)

Constructs a new JSP exception with the specified message. The message can be written to the server log and/or displayed for the user.

Parameters:
msg - a String specifying the text of the exception message

public **JspException**(java.lang.String message, java.lang.Throwable rootCause)

Constructs a new JSP exception when the JSP needs to throw an exception and include a message about the "root cause" exception that interfered with its normal operation, including a description message.

Parameters:
message - a String containing the text of the exception message

rootCause - the Throwable exception that interfered with the servlet's normal operation, making this servlet exception necessary

public **JspException**(java.lang.Throwable rootCause)

Constructs a new JSP exception when the JSP needs to throw an exception and include a message about the "root cause" exception that interfered with its normal operation. The exception's message is based on the localized message of the underlying exception.

This method calls the getLocalizedMessage method on the Throwable exception to get a localized exception message. When subclassing JspException, this method can be overridden to create an exception message designed for a specific locale.

Parameters:
rootCause - the Throwable exception that interfered with the JSP's normal operation, making the JSP exception necessary

JSP.12.4.1.2 Methods

public java.lang.Throwable **getRootCause**()

Returns the exception that caused this JSP exception.

Returns: the Throwable that caused this JSP exception

JSP.12.4.2 JspTagException

Syntax

public class JspTagException extends JspException

All Implemented Interfaces: java.io.Serializable

Description

Exception to be used by a Tag Handler to indicate some unrecoverable error. This error is to be caught by the top level of the JSP page and will result in an error page.

JSP.12.4.2.1 Constructors

public **JspTagException**()

Constructs a new JspTagException with no message.

public **JspTagException**(java.lang.String msg)

Constructs a new JspTagException with the specified message. The message can be written to the server log and/or displayed for the user.

Parameters:
msg - a String specifying the text of the exception message

public **JspTagException**(java.lang.String message,
java.lang.Throwable rootCause)

Constructs a new JspTagException when the JSP Tag needs to throw an exception and include a message about the “root cause” exception that interfered with its normal operation, including a description message.

Parameters:
message - a String containing the text of the exception message

rootCause - the Throwable exception that interfered with the JSP Tag’s normal operation, making this JSP Tag exception necessary

Since: 2.0

public **JspTagException**(java.lang.Throwable rootCause)

Constructs a new JSP Tag exception when the JSP Tag needs to throw an exception and include a message about the “root cause” exception that inter-

fered with its normal operation. The exception's message is based on the localized message of the underlying exception.

This method calls the getLocalizedMessage method on the Throwable exception to get a localized exception message. When subclassing JspTagException, this method can be overridden to create an exception message designed for a specific locale.

Parameters:
rootCause - the Throwable exception that interfered with the JSP Tag's normal operation, making the JSP Tag exception necessary

Since: 2.0

JSP.12.4.3 SkipPageException

Syntax

public class SkipPageException extends JspException

All Implemented Interfaces: java.io.Serializable

Description

Exception to indicate the calling page must cease evaluation. Thrown by a simple tag handler to indicate that the remainder of the page must not be evaluated. The result is propagated back to the pagein the case where one tag invokes another (as can be the case with tag files). The effect is similar to that of a Classic Tag Handler returning Tag.SKIP_PAGE from doEndTag(). Jsp Fragments may also throw this exception. This exception should not be thrown manually in a JSP page or tag file - the behavior is undefined. The exception is intended to be thrown inside SimpleTag handlers and in JSP fragments.

Since: 2.0

See Also: public void doTag(), public abstract void invoke(java.io.Writer out), public int doEndTag()

JSP.12.4.3.1 Constructors

public **SkipPageException**()

Creates a SkipPageException with no message.

public **SkipPageException**(java.lang.String message)

Creates a SkipPageException with the provided message.

Parameters:
message - the detail message

public **SkipPageException**(java.lang.String message,
java.lang.Throwable rootCause)

Creates a SkipPageException with the provided message and root cause.

Parameters:
message - the detail message

rootCause - the originating cause of this exception

public **SkipPageException**(java.lang.Throwable rootCause)

Creates a SkipPageException with the provided root cause.

Parameters:
rootCause - the originating cause of this exception

CHAPTER JSP.13

Tag Extension API

This chapter describes the details of tag handlers and other tag extension classes as well as methods that are available to access the Tag Library Descriptor files. This complements a previous chapter that described the Tag Library Descriptor files formats and their use in taglib directives.

This chapter includes content that is generated automatically from javadoc embedded into the actual Java classes and interfaces. This allows the creation of a single, authoritative, specification document.

Custom actions can be used by JSP authors and authoring tools to simplify writing JSP pages. A custom action can be either an empty or a non-empty action.

An empty tag has no body. There are two equivalent syntaxes, one with separate start and an end tag, and one where the start and end tags are combined. The two following examples are identical:

```
<x:foo att="myObject" />
<x:foo att="myObject" ></foo>
```

A non-empty tag has a start tag, a body, and an end tag. A prototypical example is of the form:

```
<x:foo att="myObject" >
  BODY
</x:foo/>
```

The JavaServer Pages(tm) (JSP) 1.2 specification provides a portable mechanism for the description of tag libraries containing:

- A Tag Library Descriptor (TLD)
- A number of Tag handler classes defining request-time behavior
- A number of classes defining translation-time behavior
- Additional resources used by the classes

This chapter is organized in three sections. The first section presents the basic tag handler classes. The second section describes the more complex tag handlers that need to access their body evaluation. The last section looks at translation-time issues.

JSP.13.1 Classic Tag Handlers

This section introduces the notion of a tag handler and describes the classic types of tag handler.

JSP 2.0 introduces a new type of Tag Handler called a Simple Tag Handler, which is described in a later section in this chapter. The protocol for Simple Tag handlers is much more straightforward.

Tag Handler

A tag handler is a run-time, container-managed, object that evaluates custom actions during the execution of a JSP page. A tag handler supports a protocol that allows the JSP container to provide good integration of the server-side actions within a JSP page.

A tag handler is created initially using a zero argument constructor on its corresponding class; the method java.beans.Beans.instantiate() is not used.

A tag handler has some properties that are exposed to the page as attributes on an action; these properties are managed by the JSP container (via generated code). The setter methods used to set the properties are discovered using the JavaBeans introspector machinery.

The protocol supported by a tag handler provides for passing of parameters, the evaluation and reevaluation of the body of the action, and for getting access to objects and other tag handlers in the JSP page.

A tag handler instance is responsible for processing one request at a time. It is the responsability of the JSP container to enforce this.

Additional translation time information associated with the action indicates the name of any scripting variables it may introduce, their types and their scope. At specific moments, the JSP container will automatically synchronize the PageContext information with variables in the scripting language so they can be made available directly through the scripting elements.

Properties

A tag handler has some properties. All tag handlers have a *pageContext* property for the JSP page where the tag is located, and a *parent* property for the tag handler to the closest enclosing action. Specific tag handler classes may have additional properties.

All attributes of a custom action must be JavaBeans component properties, although some properties may not be exposed as attributes. The attributes that are visible to the JSP translator are exactly those listed in the Tag Library Descriptor (TLD).

All properties of a tag handler instance exposed as attributes will be initialized by the container using the appropriate setter methods before the instance can be used to perform the action methods. It is the responsibility of the JSP container to invoke the appropriate setter methods to initialize these properties. It is the responsability of user code, be it scriptlets, JavaBeans code, or code inside custom tags, to not invoke these setter methods, as doing otherwise would interfere with the container knowledge.

The setter methods that should be used when assigning a value to an attribute of a custom action are determined by using the JavaBeans introspector on the tag handler class, then use the setter method associated with the property that has the same name as the attribute in question. An implication (unclear in the JavaBeans specification) is that there is only one setter per property.

Unspecified attributes/properties should not be set (using a setter method).

Once properly set, all properties are expected to be persistent, so that if the JSP container ascertains that a property has already been set on a given tag handler instance, it must not set it again.

The JSP container may reuse classic tag handler instances for multiple occurrences of the corresponding custom action, in the same page or in different pages, but only if the same set of attributes are used for all occurrences. If a tag handler is used for more than one occurence, the container must reset all attributes where the values differ between the custom action occurrences. Attributes with the same value in all occurrences must not be reset. If an attribute value is set as a request-time attribute value (using a scripting or an EL expression), the container must reset the attribute between all reuses of the tag handler instance.

User code can access property information and access and modify tag handler internal state starting with the first action method (doStartTag) up until the last action method (doEndTag or doFinally for tag handlers implementing TryCatch-Finally).

Tag Handler as a Container-Managed Object

Since a tag handler is a container managed object, the container needs to maintain its references; specifically, user code should not keep references to a tag handler except between the start of the first action method (doStartTag()) and the end of the last action method (doEndTag() or doFinally() for those tags that implement TryCatchFinally).

The restrictions on references to tag handler objects and on modifying attribute properties gives the JSP container substantial freedom in effectively managing tag handler objects to achieve different goals. For example, a container may implementing different pooling strategies to minimize creation cost, or may

hoist setting of properties to reduce cost when a tag handler is inside another iterative tag.

Conversions

A tag handler implements an action; the JSP container must follow the type conversions described in Section 2.13.2 when assigning values to the attributes of an action.

Empty and Non-Empty Actions

An empty action has no body; it may use one of two syntaxes: either <foo/> or <foo></foo>. Since empty actions have no body the methods related to body manipulation are not invoked. There is a mechanism in the Tag Library Descriptor to indicate that a tag can only be used to write empty actions; when used, non-empty actions using that tag will produce a translation error.

A non-empty action has a body.

The Tag Interface

A Tag handler that does not want to process its body can implement just the Tag interface. A tag handler may not want to process its body because it is an empty tag or because the body is just to be "passed through".

The Tag interface includes methods to provide page context information to the Tag Handler instance, methods to handle the life-cycle of tag handlers, and two main methods for performing actions on a tag: doStartTag() and doEndTag(). The method doStartTag() is invoked when encountering the start tag and its return value indicates whether the body (if there is any) should be skipped, or evaluated and passed through to the current response stream. The method doEndTag() is invoked when encountering the end tag; its return value indicates whether the rest of the page should continue to be evaluated or not.

If an exception is encountered during the evaluation of the body of a tag, its doEndTag method will not be evaluated. See the TryCatchFinally tag for methods that are guaranteed to be evaluated.

The IterationTag Interface

The IterationTag interface is used to repeatedly reevaluate the body of a custom action. The interface has one method: doAfterBody() which is invoked after each evaluation of the body to determine whether to reevaluate or not.

Reevaluation is requested with the value 2, which in JSP 1.1 is defined to be BodyTag.EVAL_BODY_TAG. That constant value is still kept in JSP 1.2 (for full backwards compatibility) but, to improve clarity, a new name is also available:

IterationTag.EVAL_BODY_AGAIN. To stop iterating, the returned value should be 0, which is Tag.SKIP_BODY.

The TagSupport Base Class

The TagSupport class is a base class that can be used when implementing the Tag or IterationTag interfaces.

JSP.13.1.1 JspTag

Syntax

```
public interface JspTag
```

All Known Subinterfaces: BodyTag, IterationTag, SimpleTag, Tag

Description

Serves as a base class for Tag and SimpleTag. This is mostly for organizational and type-safety purposes.

Since: 2.0

JSP.13.1.2 Tag

Syntax

```
public interface Tag extends JspTag
```

All Known Subinterfaces: BodyTag, IterationTag

All Superinterfaces: JspTag

All Known Implementing Classes: TagAdapter

Description

The interface of a classic tag handler that does not want to manipulate its body. The Tag interface defines the basic protocol between a Tag handler and JSP page implementation class. It defines the life cycle and the methods to be invoked at start and end tag.

Properties

The Tag interface specifies the setter and getter methods for the core pageContext and parent properties.

The JSP page implementation object invokes setPageContext and setParent, in that order, before invoking doStartTag() or doEndTag().

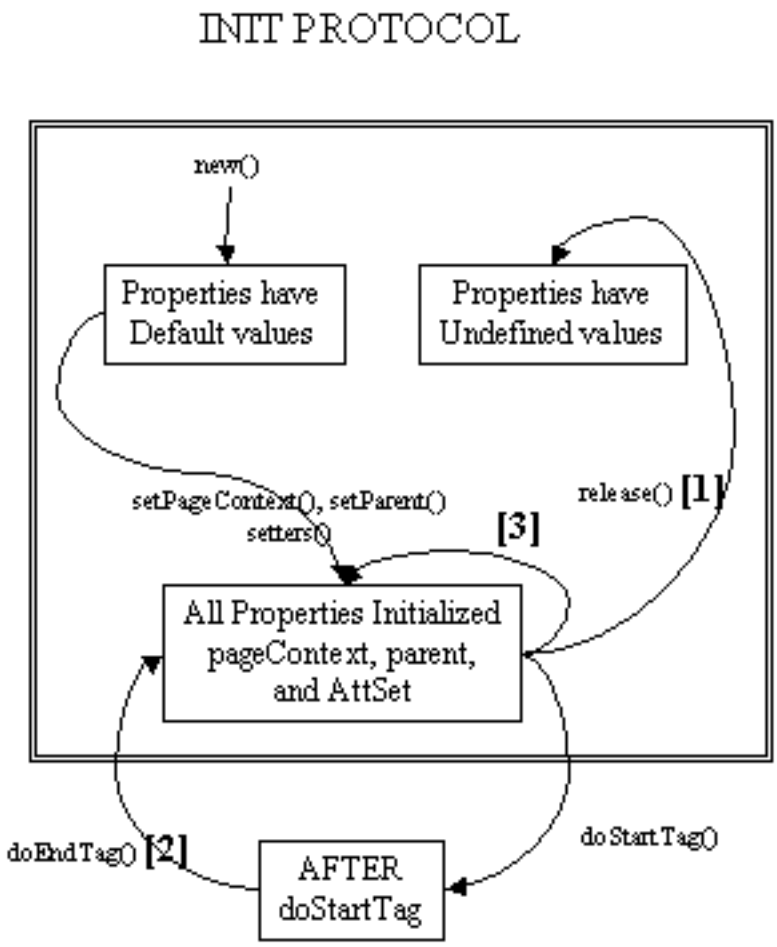
Methods

There are two main actions: doStartTag and doEndTag. Once all appropriate properties have been initialized, the doStartTag and doEndTag methods can be invoked on the tag handler. Between these invocations, the tag handler is assumed to hold a state that must be preserved. After the doEndTag invocation, the tag handler is available for further invocations (and it is expected to have retained its properties).

Lifecycle

Lifecycle details are described by the transition diagram below, with the following comments:

- [1] This transition is intended to be for releasing long-term data. no guarantees are assumed on whether any properties have been retained or not.
- [2] This transition happens if and only if the tag ends normally without raising an exception
- [3] Some setters may be called again before a tag handler is reused. For instance, setParent() is called if it's reused within the same page but at a different level, setPageContext() is called if it's used in another page, and attribute setters are called if the values differ or are expressed as request-time attribute values.
- Check the TryCatchFinally interface for additional details related to exception handling and resource management.



Once all invocations on the tag handler are completed, the release method is invoked on it. Once a release method is invoked *all* properties, including parent and pageContext, are assumed to have been reset to an unspecified value. The page compiler guarantees that release() will be invoked on the Tag handler before the handler is released to the GC.

Empty and Non-Empty Action

If the TagLibraryDescriptor file indicates that the action must always have an empty action, by an <body-content> entry of “empty”, then the doStartTag() method must return SKIP_BODY.

Otherwise, the doStartTag() method may return SKIP_BODY or EVAL_BODY_INCLUDE.

If SKIP_BODY is returned the body, if present, is not evaluated.

If EVAL_BODY_INCLUDE is returned, the body is evaluated and “passed through” to the current out.

JSP.13.1.2.1 Fields

public static final int **EVAL_BODY_INCLUDE**

Evaluate body into existing out stream. Valid return value for doStartTag.

public static final int **EVAL_PAGE**

Continue evaluating the page. Valid return value for doEndTag().

public static final int **SKIP_BODY**

Skip body evaluation. Valid return value for doStartTag and doAfterBody.

public static final int **SKIP_PAGE**

Skip the rest of the page. Valid return value for doEndTag.

JSP.13.1.2.2 Methods

public int **doEndTag**()

Process the end tag for this instance. This method is invoked by the JSP page implementation object on all Tag handlers.

This method will be called after returning from doStartTag. The body of the action may or may not have been evaluated, depending on the return value of doStartTag.

If this method returns EVAL_PAGE, the rest of the page continues to be evaluated. If this method returns SKIP_PAGE, the rest of the page is not evaluated, the request is completed, and the doEndTag() methods of enclosing tags are not invoked. If this request was forwarded or included from another page (or Servlet), only the current page evaluation is stopped.

The JSP container will resynchronize the values of any AT_BEGIN and AT_END variables (defined by the associated TagExtraInfo or TLD) after the invocation of doEndTag().

Returns: indication of whether to continue evaluating the JSP page.

Throws:
JspException - if an error occurred while processing this tag

public int **doStartTag**()

Process the start tag for this instance. This method is invoked by the JSP page implementation object.

The doStartTag method assumes that the properties pageContext and parent have been set. It also assumes that any properties exposed as attributes have been set too. When this method is invoked, the body has not yet been evaluated.

This method returns Tag.EVAL_BODY_INCLUDE or BodyTag.EVAL_BODY_BUFFERED to indicate that the body of the action should be evaluated or SKIP_BODY to indicate otherwise.

When a Tag returns EVAL_BODY_INCLUDE the result of evaluating the body (if any) is included into the current "out" JspWriter as it happens and then doEndTag() is invoked.

BodyTag.EVAL_BODY_BUFFERED is only valid if the tag handler implements BodyTag.

The JSP container will resynchronize the values of any AT_BEGIN and NESTED variables (defined by the associated TagExtraInfo or TLD) after the invocation of doStartTag(), except for a tag handler implementing BodyTag whose doStartTag() method returns BodyTag.EVAL_BODY_BUFFERED.

Returns: EVAL_BODY_INCLUDE if the tag wants to process body, SKIP_BODY if it does not want to process it.

Throws:
JspException - if an error occurred while processing this tag

See Also: BodyTag

public Tag **getParent**()

Get the parent (closest enclosing tag handler) for this tag handler.

The getParent() method can be used to navigate the nested tag handler structure at runtime for cooperation among custom actions; for example, the findAncestorWithClass() method in TagSupport provides a convenient way of doing this.

The current version of the specification only provides one formal way of indicating the observable type of a tag handler: its tag handler implementation class, described in the tag-class subelement of the tag element. This is extended in an informal manner by allowing the tag library author to indicate in the description subelement an observable type. The type should be a subtype of the tag handler implementation class or void. This addititional constraint can be exploited by a specialized container that knows about that specific tag library, as in the case of the JSP standard tag library.

Returns: the current parent, or null if none.

See Also: public static final Tag findAncestorWithClass(Tag from, java.lang.Class klass)

public void **release**()

Called on a Tag handler to release state. The page compiler guarantees that JSP page implementation objects will invoke this method on all tag handlers, but there may be multiple invocations on doStartTag and doEndTag in between.

public void **setPageContext**(PageContext pc)

Set the current page context. This method is invoked by the JSP page implementation object prior to doStartTag().

This value is *not* reset by doEndTag() and must be explicitly reset by a page implementation if it changes between calls to doStartTag().

Parameters:
pc - The page context for this tag handler.

public void **setParent**(Tag t)

Set the parent (closest enclosing tag handler) of this tag handler. Invoked by the JSP page implementation object prior to doStartTag().

This value is *not* reset by doEndTag() and must be explicitly reset by a page implementation.

Parameters:
t - The parent tag, or null.

JSP.13.1.3 IterationTag

Syntax

public interface IterationTag extends Tag

All Known Subinterfaces: BodyTag

All Superinterfaces: JspTag, Tag

All Known Implementing Classes: TagSupport

Description

The IterationTag interface extends Tag by defining one additional method that controls the reevaluation of its body.

A tag handler that implements IterationTag is treated as one that implements Tag regarding the doStartTag() and doEndTag() methods. IterationTag provides a new method: doAfterBody().

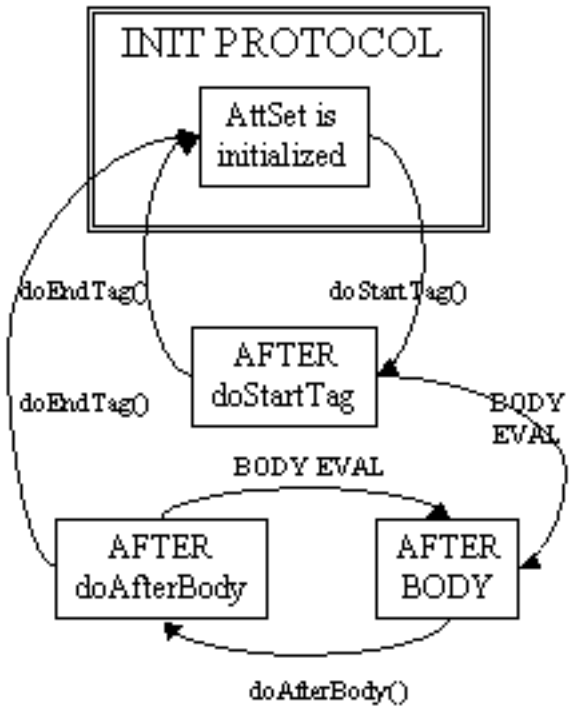
The doAfterBody() method is invoked after every body evaluation to control whether the body will be reevaluated or not. If doAfterBody() returns IterationTag.EVAL_BODY_AGAIN, then the body will be reevaluated. If doAfterBody() returns Tag.SKIP_BODY, then the body will be skipped and doEndTag() will be evaluated instead.

Properties There are no new properties in addition to those in Tag.

Methods There is one new methods: doAfterBody().

Lifecycle

Lifecycle details are described by the transition diagram below. Exceptions that are thrown during the computation of doStartTag(), BODY and doAfterBody() interrupt the execution sequence and are propagated up the stack, unless the tag handler implements the TryCatchFinally interface; see that interface for details.



Empty and Non-Empty Action

If the TagLibraryDescriptor file indicates that the action must always have an empty element body, by a <body-content> entry of “empty”, then the doStartTag() method must return SKIP_BODY.

Note that which methods are invoked after the doStartTag() depends on both the return value and on if the custom action element is empty or not in the JSP page, not on how it’s declared in the TLD.

If SKIP_BODY is returned the body is not evaluated, and then doEndTag() is invoked.

If EVAL_BODY_INCLUDE is returned, and the custom action element is not empty, the body is evaluated and “passed through” to the current out, then doAfterBody() is invoked and, after zero or more iterations, doEndTag() is invoked.

JSP.13.1.3.1 Fields

public static final int **EVAL_BODY_AGAIN**

Request the reevaluation of some body. Returned from doAfterBody. For compatibility with JSP 1.1, the value is carefully selected to be the same as the, now deprecated, BodyTag.EVAL_BODY_TAG,

JSP.13.1.3.2 Methods

public int **doAfterBody**()

Process body (re)evaluation. This method is invoked by the JSP Page implementation object after every evaluation of the body into the BodyEvaluation object. The method is not invoked if there is no body evaluation.

If doAfterBody returns EVAL_BODY_AGAIN, a new evaluation of the body will happen (followed by another invocation of doAfterBody). If doAfterBody returns SKIP_BODY, no more body evaluations will occur, and the doEndTag method will be invoked.

If this tag handler implements BodyTag and doAfterBody returns SKIP_BODY, the value of out will be restored using the popBody method in pageContext prior to invoking doEndTag.

The method re-invocations may be lead to different actions because there might have been some changes to shared state, or because of external computation.

The JSP container will resynchronize the values of any AT_BEGIN and NESTED variables (defined by the associated TagExtraInfo or TLD) after the invocation of doAfterBody().

Returns: whether additional evaluations of the body are desired

Throws:
JspException - if an error occurred while processing this tag

JSP.13.1.4 TryCatchFinally

Syntax

public interface TryCatchFinally

Description

The auxiliary interface of a Tag, IterationTag or BodyTag tag handler that wants additional hooks for managing resources.

This interface provides two new methods: doCatch(Throwable) and doFinally(). The prototypical invocation is as follows:

```
h = get a Tag();  // get a tag handler, perhaps from pool
h.setPageContext(pc);  // initialize as desired
h.setParent(null);
h.setFoo("foo");

// tag invocation protocol; see Tag.java
try {
  doStartTag()...
  ....
  doEndTag()...
} catch (Throwable t) {
  // react to exceptional condition
  h.doCatch(t);
} finally {
  // restore data invariants and release per-invocation resources
  h.doFinally();
}

... other invocations perhaps with some new setters
...
h.release();  // release long-term resources
```

JSP.13.1.4.1 Methods

public void **doCatch**(java.lang.Throwable t)

Invoked if a Throwable occurs while evaluating the BODY inside a tag or in any of the following methods: Tag.doStartTag(), Tag.doEndTag(), IterationTag.doAfterBody() and BodyTag.doInitBody().

This method is not invoked if the Throwable occurs during one of the setter methods.

This method may throw an exception (the same or a new one) that will be propagated further up the nest chain. If an exception is thrown, doFinally() will be invoked.

This method is intended to be used to respond to an exceptional condition.

Parameters:
t - The throwable exception navigating through this tag.

Throws:
Throwable - if the exception is to be rethrown further up the nest chain.

public void **doFinally**()

Invoked in all cases after doEndTag() for any class implementing Tag, IterationTag or BodyTag. This method is invoked even if an exception has occurred in the BODY of the tag, or in any of the following methods: Tag.doStartTag(), Tag.doEndTag(), IterationTag.doAfterBody() and BodyTag.doInitBody().

This method is not invoked if the Throwable occurs during one of the setter methods.

This method should not throw an Exception.

This method is intended to maintain per-invocation data integrity and resource management actions.

JSP.13.1.5 TagSupport

Syntax

public class TagSupport implements IterationTag, java.io.Serializable

Direct Known Subclasses: BodyTagSupport

All Implemented Interfaces: IterationTag, JspTag, java.io.Serializable, Tag

Description

A base class for defining new tag handlers implementing Tag.

The TagSupport class is a utility class intended to be used as the base class for new tag handlers. The TagSupport class implements the Tag and IterationTag interfaces and adds additional convenience methods including getter methods for the properties in Tag. TagSupport has one static method that is included to facilitate coordination among cooperating tags.

Many tag handlers will extend TagSupport and only redefine a few methods.

JSP.13.1.5.1 Fields

protected java.lang.String **id**

The value of the id attribute of this tag; or null.

protected PageContext **pageContext**

The PageContext.

JSP.13.1.5.2 Constructors

public **TagSupport**()

Default constructor, all subclasses are required to define only a public constructor with the same signature, and to call the superclass constructor. This constructor is called by the code generated by the JSP translator.

JSP.13.1.5.3 Methods

public int **doAfterBody**()

Default processing for a body.

Returns: SKIP_BODY

Throws:
JspException - if an error occurs while processing this tag

See Also: public int doAfterBody()

public int **doEndTag**()

Default processing of the end tag returning EVAL_PAGE.

Returns: EVAL_PAGE

Throws:
JspException - if an error occurs while processing this tag

See Also: public int doEndTag()

public int **doStartTag**()

Default processing of the start tag, returning SKIP_BODY.

Returns: SKIP_BODY

Throws:
JspException - if an error occurs while processing this tag

See Also: public int doStartTag()

public static final Tag **findAncestorWithClass**(Tag from, java.lang.Class klass)

Find the instance of a given class type that is closest to a given instance. This method uses the getParent method from the Tag interface. This method is used for coordination among cooperating tags.

The current version of the specification only provides one formal way of indicating the observable type of a tag handler: its tag handler implementation class, described in the tag-class subelement of the tag element. This is extended in an informal manner by allowing the tag library author to indicate in the description subelement an observable type. The type should be a subtype of the tag handler implementation class or void. This addititional constraint can be exploited by a specialized container that knows about that specific tag library, as in the case of the JSP standard tag library.

When a tag library author provides information on the observable type of a tag handler, client programmatic code should adhere to that constraint. Specifically, the Class passed to findAncestorWithClass should be a subtype of the observable type.

Parameters:
from - The instance from where to start looking.

klass - The subclass of Tag or interface to be matched

Returns: the nearest ancestor that implements the interface or is an instance of the class specified

public java.lang.String **getId**()

The value of the id attribute of this tag; or null.

Returns: the value of the id attribute, or null

public Tag **getParent**()

The Tag instance most closely enclosing this tag instance.

Returns: the parent tag instance or null

See Also: public Tag getParent()

public java.lang.Object **getValue**(java.lang.String k)

Get a the value associated with a key.

Parameters:

k - The string key.

Returns: The value associated with the key, or null.

public java.util.Enumeration **getValues**()

Enumerate the keys for the values kept by this tag handler.

Returns: An enumeration of all the keys for the values set, or null or an empty Enumeration if no values have been set.

public void **release**()

Release state.

See Also: public void release()

public void **removeValue**(java.lang.String k)

Remove a value associated with a key.

Parameters:
k - The string key.

public void **setId**(java.lang.String id)

Set the id attribute for this tag.

Parameters:
id - The String for the id.

public void **setPageContext**(PageContext pageContext)

Set the page context.

Parameters:
pageContext - The PageContext.

See Also: public void setPageContext(PageContext pc)

public void **setParent**(Tag t)

Set the nesting tag of this tag.

Parameters:
t - The parent Tag.

See Also: public void setParent(Tag t)

public void **setValue**(java.lang.String k, java.lang.Object o)

Associate a value with a String key.

Parameters:
k - The key String.

o - The value to associate.

JSP.13.2 Tag Handlers that want Access to their Body Content

The evaluation of a body is delivered into a BodyContent object. This is then made available to tag handlers that implement the BodyTag interface. The BodyTagSupport class provides a useful base class to simplify writing these handlers.

If a Tag handler wants to have access to the content of its body then it must implement the BodyTag interface. This interface extends IterationTag, provides two additional methods setBodyContent(BodyContent) and doInitBody() and refers to an object of type BodyContent.

A BodyContent is a subclass of JspWriter that has a few additional methods to convert its contents into a String, insert the contents into another JspWriter, to get a Reader into its contents, and to clear the contents. Its semantics also assure that buffer size will never be exceeded.

The JSP page implementation will create a BodyContent if the doStartTag() method returns a EVAL_BODY_BUFFERED. This object will be passed to doInitBody(); then the body of the tag will be evaluated, and *during that evaluation* ***out will be bound to the BodyContent just passed to the BodyTag handler***. Then doAfterBody() will be evaluated. If that method returns SKIP_BODY, no more evaluations of the body will be done; if the method returns EVAL_BODY_AGAIN, then the body will be evaluated, and doAfterBody() will be invoked again.

The content of a BodyContent instance remains available until after the invocation of its associated doEndTag() method.

A common use of the BodyContent is to extract its contents into a String and then use the String as a value for some operation. Another common use is to take its contents and push it into the out Stream that was valid when the start tag was encountered (that is available from the PageContext object passed to the handler in setPageContext).

JSP.13.2.1 BodyContent

Syntax

```
public abstract class BodyContent extends JspWriter
```

Description

An encapsulation of the evaluation of the body of an action so it is available to a tag handler. BodyContent is a subclass of JspWriter.

Note that the content of BodyContent is the result of evaluation, so it will not contain actions and the like, but the result of their invocation.

BodyContent has methods to convert its contents into a String, to read its contents, and to clear the contents.

The buffer size of a BodyContent object is unbounded. A BodyContent object cannot be in autoFlush mode. It is not possible to invoke flush on a BodyContent object, as there is no backing stream.

Instances of BodyContent are created by invoking the pushBody and popBody methods of the PageContext class. A BodyContent is enclosed within another JspWriter (maybe another BodyContent object) following the structure of their associated actions.

A BodyContent is made available to a BodyTag through a setBodyContent() call. The tag handler can use the object until after the call to doEndTag().

JSP.13.2.1.1 Constructors

protected **BodyContent**(JspWriter e)

Protected constructor. Unbounded buffer, no autoflushing.

Parameters:
e - the enclosing JspWriter

JSP.13.2.1.2 Methods

public void **clearBody**()

Clear the body without throwing any exceptions.

public void **flush**()

Redefined flush() so it is not legal.

It is not valid to flush a BodyContent because there is no backing stream behind it.

Overrides: public abstract void flush() in class JspWriter

Throws:
IOException - always thrown

public JspWriter **getEnclosingWriter**()

Get the enclosing JspWriter.

Returns: the enclosing JspWriter passed at construction time

public abstract java.io.Reader **getReader**()

Return the value of this BodyContent as a Reader.

Returns: the value of this BodyContent as a Reader

public abstract java.lang.String **getString**()

Return the value of the BodyContent as a String.

Returns: the value of the BodyContent as a String

public abstract void **writeOut**(java.io.Writer out)

Write the contents of this BodyContent into a Writer. Subclasses may optimize common invocation patterns.

Parameters:
out - The writer into which to place the contents of this body evaluation

Throws:
IOException - if an I/O error occurred while writing the contents of this BodyContent to the given Writer

JSP.13.2.2 BodyTag

Syntax

public interface BodyTag extends IterationTag

All Superinterfaces: IterationTag, JspTag, Tag

All Known Implementing Classes: BodyTagSupport

Description

The BodyTag interface extends IterationTag by defining additional methods that let a tag handler manipulate the content of evaluating its body.

It is the responsibility of the tag handler to manipulate the body content. For example the tag handler may take the body content, convert it into a String using the bodyContent.getString method and then use it. Or the tag handler may take the body content and write it out into its enclosing JspWriter using the bodyContent.writeOut method.

A tag handler that implements BodyTag is treated as one that implements IterationTag, except that the doStartTag method can return SKIP_BODY, EVAL_BODY_INCLUDE or EVAL_BODY_BUFFERED.

If EVAL_BODY_INCLUDE is returned, then evaluation happens as in IterationTag.

If EVAL_BODY_BUFFERED is returned, then a BodyContent object will be created (by code generated by the JSP compiler) to capture the body evaluation. The code generated by the JSP compiler obtains the BodyContent object by calling the pushBody method of the current pageContext, which additionally has the effect of saving the previous out value. The page compiler returns this object by calling the popBody method of the PageContext class; the call also restores the value of out.

The interface provides one new property with a setter method and one new action method.

Properties

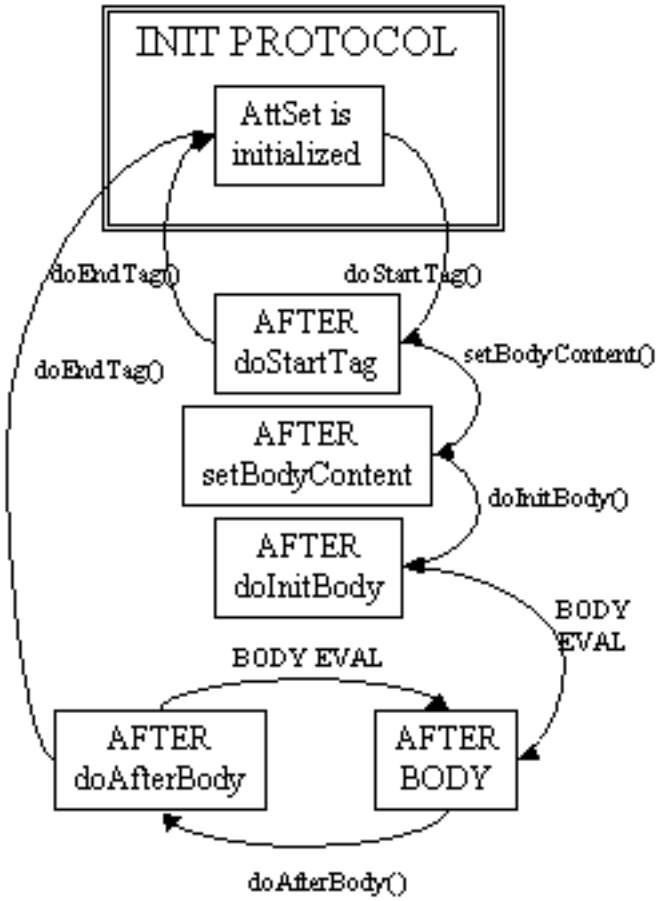
There is a new property: bodyContent, to contain the BodyContent object, where the JSP Page implementation object will place the evaluation (and reevaluation, if appropriate) of the body. The setter method (setBodyContent) will only be invoked if doStartTag() returns EVAL_BODY_BUFFERED and the corresponding action element does not have an empty body.

Methods

In addition to the setter method for the bodyContent property, there is a new action method: doInitBody(), which is invoked right after setBodyContent() and before the body evaluation. This method is only invoked if doStartTag() returns EVAL_BODY_BUFFERED.

Lifecycle

Lifecycle details are described by the transition diagram below. Exceptions that are thrown during the computation of doStartTag(), setBodyContent(), doInitBody(), BODY, doAfterBody() interrupt the execution sequence and are propagated up the stack, unless the tag handler implements the TryCatchFinally interface; see that interface for details.



Empty and Non-Empty Action

If the TagLibraryDescriptor file indicates that the action must always have an empty element body, by an <body-content> entry of “empty”, then the doStart-Tag() method must return SKIP_BODY. Otherwise, the doStartTag() method may return SKIP_BODY, EVAL_BODY_INCLUDE, or EVAL_BODY_BUFFERED.

Note that which methods are invoked after the doStartTag() depends on both the return value and on if the custom action element is empty or not in the JSP page, not how it’s declared in the TLD.

If SKIP_BODY is returned the body is not evaluated, and doEndTag() is invoked.

If EVAL_BODY_INCLUDE is returned, and the custom action element is not empty, setBodyContent() is not invoked, doInitBody() is not invoked, the body is

evaluated and “passed through” to the current out, doAfterBody() is invoked and then, after zero or more iterations, doEndTag() is invoked. If the custom action element is empty, only doStart() and doEndTag() are invoked.

If EVAL_BODY_BUFFERED is returned, and the custom action element is not empty, setBodyContent() is invoked, doInitBody() is invoked, the body is evaluated, doAfterBody() is invoked, and then, after zero or more iterations, doEndTag() is invoked. If the custom action element is empty, only doStart() and doEndTag() are invoked.

JSP.13.2.2.1 Fields

public static final int **EVAL_BODY_BUFFERED**

Request the creation of new buffer, a BodyContent on which to evaluate the body of this tag. Returned from doStartTag when it implements BodyTag. This is an illegal return value for doStartTag when the class does not implement BodyTag.

public static final int **EVAL_BODY_TAG**

Deprecated. As of Java JSP API 1.2, use BodyTag.EVAL_BODY_BUFFERED or IterationTag.EVAL_BODY_AGAIN.

Deprecated constant that has the same value as EVAL_BODY_BUFFERED and EVAL_BODY_AGAIN. This name has been marked as deprecated to encourage the use of the two different terms, which are much more descriptive.

JSP.13.2.2.2 Methods

public void **doInitBody**()

Prepare for evaluation of the body. This method is invoked by the JSP page implementation object after setBodyContent and before the first time the body is to be evaluated. This method will not be invoked for empty tags or for non-empty tags whose doStartTag() method returns SKIP_BODY or EVAL_BODY_INCLUDE.

The JSP container will resynchronize the values of any AT_BEGIN and NESTED variables (defined by the associated TagExtraInfo or TLD) after the invocation of doInitBody().

Throws:
JspException - if an error occurred while processing this tag

See Also: public int doAfterBody()

public void **setBodyContent**(BodyContent b)

Set the bodyContent property. This method is invoked by the JSP page implementation object at most once per action invocation. This method will be invoked before doInitBody. This method will not be invoked for empty tags or for non-empty tags whose doStartTag() method returns SKIP_BODY or EVAL_BODY_INCLUDE.

When setBodyContent is invoked, the value of the implicit object out has already been changed in the pageContext object. The BodyContent object passed will have not data on it but may have been reused (and cleared) from some previous invocation.

The BodyContent object is available and with the appropriate content until after the invocation of the doEndTag method, at which case it may be reused.

Parameters:
b - the BodyContent

See Also: public void doInitBody(), public int doAfterBody()

JSP.13.2.3 BodyTagSupport

Syntax

public class BodyTagSupport extends TagSupport implements BodyTag

All Implemented Interfaces: BodyTag, IterationTag, JspTag, java.io.Serializable, Tag

Description

A base class for defining tag handlers implementing BodyTag.

The BodyTagSupport class implements the BodyTag interface and adds additional convenience methods including getter methods for the bodyContent property and methods to get at the previous out JspWriter.

Many tag handlers will extend BodyTagSupport and only redefine a few methods.

JSP.13.2.3.1 Fields

protected BodyContent **bodyContent**

The current BodyContent for this BodyTag.

JSP.13.2.3.2 Constructors

public **BodyTagSupport**()

Default constructor, all subclasses are required to only define a public constructor with the same signature, and to call the superclass constructor. This constructor is called by the code generated by the JSP translator.

JSP.13.2.3.3 Methods

public int **doAfterBody**()

After the body evaluation: do not reevaluate and continue with the page. By default nothing is done with the bodyContent data (if any).

Overrides: public int doAfterBody() in class TagSupport

Returns: SKIP_BODY

Throws:
JspException - if an error occurred while processing this tag

See Also: public void doInitBody(), public int doAfterBody()

public int **doEndTag**()

Default processing of the end tag returning EVAL_PAGE.

Overrides: public int doEndTag() in class TagSupport

Returns: EVAL_PAGE

Throws:
JspException - if an error occurred while processing this tag

See Also: public int doEndTag()

public void **doInitBody**()

Prepare for evaluation of the body just before the first body evaluation: no action.

Throws:
JspException - if an error occurred while processing this tag

See Also: public void setBodyContent(BodyContent b), public int doAfterBody(), public void doInitBody()

public int **doStartTag**()

Default processing of the start tag returning EVAL_BODY_BUFFERED.

Overrides: public int doStartTag() in class TagSupport

Returns: EVAL_BODY_BUFFERED

Throws:
JspException - if an error occurred while processing this tag

See Also: public int doStartTag()

public BodyContent **getBodyContent**()

Get current bodyContent.

Returns: the body content.

public JspWriter **getPreviousOut**()

Get surrounding out JspWriter.

Returns: the enclosing JspWriter, from the bodyContent.

public void **release**()

Release state.

Overrides: public void release() in class TagSupport

See Also: public void release()

public void **setBodyContent**(BodyContent b)

Prepare for evaluation of the body: stash the bodyContent away.

Parameters:
b - the BodyContent

See Also: public int doAfterBody(), public void doInitBody(), public void setBodyContent(BodyContent b)

JSP.13.3 Dynamic Attributes

Any tag handler can optionally extend the DynamicAttributes interface to indicate that it supports dynamic attributes. In addition to implementing the DynamicAttributes interface, tag handlers that support dynamic attributes must declare that they do so in the Tag Library Descriptor.

The TLD is what ultimately determines whether a tag handler accepts dynamic attributes or not. If a tag handler declares that it supports dynamic attributes in the TLD but it does not implement the DynamicAttributes interface, the tag handler must be considered invalid by the container.

If the dynamic-attributes element for a tag being invoked contains the value “true”, the following requirements apply:

- For each attribute specified in the tag invocation that does not have a corresponding attribute element in the TLD for this tag, a call must be made to setDynamicAttribute(), passing in the namespace of the attribute (or null if the attribute does not have a namespace or prefix), the name of the attribute without the namespace prefix, and the final value of the attribute.
- Dynamic attributes must be considered to accept request-time expression values.

- Dynamic attributes must be treated as though they were of type java.lang.Object
- The JSP container must recognize dynamic attributes that are passed to the tag handler using the <jsp:attribute> standard action.
- If the setDynamicAttribute() method throws JspException, the doStartTag() or doTag() method is not invoked for this tag, and the exception must be treated in the same manner as if it came from a regular attribute setter method.
- For a JSP document in either standard or XML syntax, If a dynamic attribute has a prefix that doesn't map to a namespace, a translation error must occur. In standard syntax, only namespaces defined using taglib directives are recognized.

In the following example, assume attributes a and b are declared using the attribute element in the TLD, attributes d1 and d2 are not declared, and the dynamic-attributes element is set to "true". The attributes are set using the calls:

- setA("1"),
- setDynamicAttribute(null, "d1", "2"),
- setDynamicAttribute("http://www.foo.com/jsp/taglib/mytag.tld", "d2", "3"),
- setB("4"),
- setDynamicAttribute(null, "d3", "5"), and
- setDynamicAttribute("http://www.foo.com/jsp/taglib/mytag.tld", "d4", "6").

```
<jsp:root xmlns:mytag="http://www.foo.com/jsp/taglib/mytag.tld" version="2.0">
  <mytag:invokeDynamic a="1" d1="2" mytag:d2="3">
    <jsp:attribute name="b">4</jsp:attribute>
    <jsp:attribute name="d3">5</jsp:attribute>
    <jsp:attribute name="mytag:d4">6</jsp:attribute>
  </mytag:invokeDynamic>
</jsp:root>
```

JSP.13.3.1 DynamicAttributes

Syntax

public interface DynamicAttributes

Description

For a tag to declare that it accepts dynamic attributes, it must implement this interface. The entry for the tag in the Tag Library Descriptor must also be configured to indicate dynamic attributes are accepted.

For any attribute that is not declared in the Tag Library Descriptor for this tag, instead of getting an error at translation time, the setDynamicAttribute() method is called, with the name and value of the attribute. It is the responsibility of the tag to remember the names and values of the dynamic attributes.

Since: 2.0

JSP.13.3.1.1 Methods

public void **setDynamicAttribute**(java.lang.String uri, java.lang.String localName, java.lang.Object value)

Called when a tag declared to accept dynamic attributes is passed an attribute that is not declared in the Tag Library Descriptor.

Parameters:
uri - the namespace of the attribute, or null if in the default namespace.

localName - the name of the attribute being set.

value - the value of the attribute

Throws:
JspException - if the tag handler wishes to signal that it does not accept the given attribute. The container must not call doStartTag() or doTag() for this tag.

JSP.13.4 Annotated Tag Handler Management Example

Below is a somewhat complete example of the way one JSP container could choose to do some tag handler management. There are many other strategies that could be followed, with different pay offs.

The example is as below. In this example, we are assuming that x:iterate is an iterative tag, while x:doit and x:foobar are simple tag. We will also assume that x:iterate and x:foobar implement the TryCatchFinally interface, while x:doit does not.

```
<x:iterate src="foo">
  <x:doit att1="one" att2="<%= 1 + 1 %>" />
  <x:foobar />
  <x:doit att1="one" att2="<%= 2 + 2 %>" />
</x:iterate>
<x:doit att1="one" att2="<%= 3 + 3 %>" />
```

The particular code shown below assumes there is some pool of tag handlers that are managed (details not described, although pool managing is simpler when there are no optional attributes), and attemps to reuse tag handlers if possible. The code also "hoists" setting of properties to reduce the cost when appropriate, e.g. inside an iteration.

```
boolean b1, b2;
IterationTag i; // for x:iterate
Tag d; // for x:doit
Tag d; // for x:foobar
page: // label to end of page...
// initialize iteration tag
i = get tag from pool or new();
i.setPageContext(pc);
i.setParent(null);
i.setSrc("foo");
// x:iterate implements TryCatchFinally
try {
    if ((b1 = i.doStartTag()) == EVAL_BODY_INCLUDE) {
        // initialize doit tag
        // code has been moved out of the loop for show
        d = get tag from pool or new();
        d.setPageContext(pc);
        d.setParent(i);
        d.setAtt1("one");
    loop:
        while (1) do {
            // I'm ignoring newlines...
            // two invocations, fused together
            // first invocation of x:doit
            d.setAtt2(1+1);
            if ((b2 = d.doStartTag()) == EVAL_BODY_INCLUDE) {
                // nothing
            } else if (b2 != SKIP_BODY) {
                // Q? protocol error ...
            }
            if ((b2 = d.doEndTag()) == SKIP_PAGE) {
                break page;  // be done with it.
            } else if (b2 != EVAL_PAGE) {
                // Q? protocol error
            }
    // x:foobar invocation
            f = get tag from pool or new();
            f.setPageContext(pc);
            f.setParent(i);
            // x:foobar implements TryCatchFinally
            try {

                if ((b2 = f.doStartTag()) == EVAL_BODY_INCLUDE) {
                    // nothing
                } else if (b2 != SKIP_BODY) {
                    // Q? protocol error
                }
                if ((b2 = f.doEndTag()) == SKIP_PAGE) {
                    break page;  // be done with it.
                } else if (b2 != EVAL_PAGE) {
                    // Q? protocol error
                }
            } catch (Throwable t) {
```

```
                f.doCatch(t); // caught, may been rethrown!
            } finally {
                f.doFinally();
            }
            // put f back to pool

            // second invocation of x:doit
            d.setAtt2(2+2);
            if ((b2 = d.doStartTag()) == EVAL_BODY_INCLUDE) {
                // nothing
            } else if (b2 != SKIP_BODY) {
                // Q? protocol error
            }
            if ((b2 = d.doEndTag()) == SKIP_PAGE) {
                break page;  // be done with it.
            } else if (b2 != EVAL_PAGE) {
                // Q? protocol error
            }
            if ((b2 = i.doAfterBody()) == EVAL_BODY_AGAIN) {
                break loop;
            } else if (b2 != SKIP_BODY) {
                // Q? protocol error
            }
        // loop
        }
    } else if (b1 != SKIP_BODY) {
        // Q? protocol error
    }
    // tail end of the IteratorTag ...
    if ((b1 = i.doEndTag()) == SKIP_PAGE) {
        break page;   // be done with it.
    } else if (b1 != EVAL_PAGE) {
        // Q? protocol error
    }

    // third invocation
    // this tag handler could be reused from the previous ones.
    d = get tag from pool or new();
    d.setPageContext(pc);
    d.setParent(null);
    d.setAtt1("one");
    d.setAtt2(3+3);
    if ((b1 = d.doStartTag()) == EVAL_BODY_INCLUDE) {
        // nothing
    } else if (b1 != SKIP_BODY) {
        // Q? protocol error
    }
    if ((b1 = d.doEndTag()) == SKIP_PAGE) {
        break page;  // be done with it.
    } else if (b1 != EVAL_PAGE) {
        // Q? protocol error
    }
} catch (Throwable t) {
```

```
    i.doCatch(t); // caught, may been rethrown!
} finally {
    i.doFinally();
}
```

JSP.13.5 Cooperating Actions

Actions can cooperate with other actions and with scripting code in a number of ways.

PageContext

Often two actions in a JSP page will want to cooperate, perhaps by one action creating some server-side object that needs to be access by another. One mechanism for doing this is by giving the object a name within the JSP page; the first action will create the object and associate the name to it while the second action will use the name to retrieve the object.

For example, in the following JSP segment the foo action might create a server-side object and give it the name “myObject”. Then the bar action might access that server-side object and take some action.

```
<x:foo id=“myObject” />
<x:bar ref=“myObjet” />
```

In a JSP implementation, the mapping “name”->value is kept by the implicit object pageContext. This object is passed around through the Tag handler instances so it can be used to communicate information: all it is needed is to know the name under which the information is stored into the pageContext.

The Runtime Stack

An alternative to explicit communication of information through a named object is implicit coordination based on syntactic scoping.

For example, in the following JSP segment the foo action might create a server-side object; later the nested bar action might access that server-side object. The object is not named within the pageContext: it is found because the specific foo element is the closest enclosing instance of a known element type.

```
<foo>
  <bar/>
</foo>
```

This functionality is supported through the TagSupport.findAncestorWith-Class(Tag, Class), which uses a reference to parent tag kept by each Tag instance, which effectively provides a run-time execution stack.

JSP.13.6 Simple Tag Handlers

This section presents the API to implement Simple Tag Handlers and JSP Fragments. Simple Tag Handlers present a much simpler invocation protocol than do Classic Tag Handlers.

The Tag Library Descriptor maps tag library declarations to their physical underlying implementations. A Simple Tag Handler is represented in Java by a class which implements the SimpleTag interface.

Unlike classic tag handlers, the SimpleTag interface does not extend Tag. Instead of supporting doStartTag() and doEndTag(), the SimpleTag interface provides a simple doTag() method, which is called once and only once for any given tag invocation. All tag logic, iteration, body evaluations, etc. are to be performed in this single method. Thus, simple tag handlers have the equivalent power of BodyTag, but with a much simpler lifecycle and interface.

To support body content, the setJspBody() method is provided. The container invokes the setJspBody() method with a JspFragment object encapsulating the body of the tag. The tag handler implementation can call invoke() on that fragment to evaluate the body. The SimpleTagSupport convenience class provides getJspBody() and other useful methods to make this even easier.

Lifecycle of Simple Tag Handlers

This section describes the lifecycle of simple tag handlers, from creation to invocation. For all semantics left unspecified by this section, the semantics default to that of a classic tag handler.

When a simple tag handler is invoked, the following steps occur (in order):

1. Simple tag handlers are created initially using a zero argument constructor on the corresponding implementation class. Unlike classic tag handlers, this instance must never be pooled by the container. A new instance must be created for each tag invocation.
2. The setJspContext() and setParent() methods are invoked on the tag handler. The setParent() method need not be called if the value being passed in is null. In the case of tag files, a JspContext wrapper is created so that the tag file can appear to have its own page scope. Calling getJspContext() must return the wrapped JspContext.
3. The attributes specified as XML element attributes (if any) are evaluated next, in the order in which they are declared, according to the following rules (referred to as “evaluating an XML element attribute” below). The appropriate bean property setter is invoked for each. If no setter is defined for the specified attribute but the tag accepts dynamic attributes, the setDynamicAttribute() method is invoked as the setter.

•If the attribute is a scripting expression (e.g. “<%= 1+1 %>” in JSP syntax, or “%= 1+1 %” in XML syntax), the expression is evaluated, and the result is converted as per the rules in “Type Conversions”, and passed to the setter.
•Otherwise, if the attribute contains any Expression Language expressions (e.g. “Hello ${name}”), the expression is evaluated, and the result is converted and passed to the setter.
•Otherwise, the attribute value is taken verbatim, converted, and passed to the setter.

4. The value for each <jsp:attribute> element is evaluated, and the corresponding bean property setter methods are invoked for each, in the order in which they appear in the body of the tag. If no setter is defined for the specified attribute but the tag accepts dynamic attributes, the setDynamicAttribute() method is invoked as the setter.

   •Otherwise, if the attribute is not of type JspFragment, the container evaluates the body of the <jsp:attribute> element. This evaluation can be done in a container-specific manner. Container implementors should note that in the process of evaluating this body, other custom actions may be invoked.
   •Otherwise, if the attribute is of type JspFragment, an instance of a JspFragment object is created and passed in.

5. The value for the body of the tag is determined, and if a body exists the setJspBody() method is called on the tag handler.

   •If the tag is declared to have a body-content of “empty” or no body or an empty body is passed for this invocation, then setJspBody() is not called.
   •Otherwise, the body of the tag is either the body of the <jsp:body> element, or the body of the custom action invocation if no <jsp:body> or <jsp:attribute> elements are present. In this case, an instance of a JspFragment object is created as per the lifecycle described in the JSP Fragments section and it is passed to the setter. If the tag is declared to have a body-content of “tagdependent” the JspFragment must echo the body’s contents verbatim. Otherwise, if the tag is declared to have a body-content of type “scriptless”, the JspFragment must evaluate the body’s contents as a JSP scriptless body.

6. The doTag() method is invoked.

7. The implementation of doTag() performs its function, potentially calling other tag handlers (if the tag handler is implemented as a tag file) and invoking fragments.

8. The doTag() method returns, and the tag handler instance is discarded. If SkipPageException is thrown, the rest of the page is not evaluated and the request is completed. If this request was forwarded or included from another page (or Servlet), only the current page evaluation stops.

9. For each tag scripting variable declared with scopes AT_BEGIN or AT_END, the appropriate scripting variables and scoped attributes are declared, as with classic

tag handlers.

JSP.13.6.1 SimpleTag

Syntax

public interface SimpleTag extends JspTag

All Superinterfaces: JspTag

All Known Implementing Classes: SimpleTagSupport

Description

Interface for defining Simple Tag Handlers.

Simple Tag Handlers differ from Classic Tag Handlers in that instead of supporting doStartTag() and doEndTag(), the SimpleTag interface provides a simple doTag() method, which is called once and only once for any given tag invocation. All tag logic, iteration, body evaluations, etc. are to be performed in this single method. Thus, simple tag handlers have the equivalent power of BodyTag, but with a much simpler lifecycle and interface.

To support body content, the setJspBody() method is provided. The container invokes the setJspBody() method with a JspFragment object encapsulating the body of the tag. The tag handler implementation can call invoke() on that fragment to evaluate the body as many times as it needs.

A SimpleTag handler must have a public no-args constructor. Most SimpleTag handlers should extend SimpleTagSupport.

Lifecycle

The following is a non-normative, brief overview of the SimpleTag lifecycle. Refer to the JSP Specification for details.

1. A new tag handler instance is created each time by the container by calling the provided zero-args constructor. Unlike classic tag handlers, simple tag handlers are never cached and reused by the JSP container.
2. The setJspContext() and setParent() methods are called by the container. The setParent() method is only called if the element is nested within another tag invocation.
3. The setters for each attribute defined for this tag are called by the container.
4. If a body exists, the setJspBody() method is called by the container to set the body of this tag, as a JspFragment. If the action element is empty in the page, this

method is not called at all.

5. The doTag() method is called by the container. All tag logic, iteration, body evaluations, etc. occur in this method.
6. The doTag() method returns and all variables are synchronized.

Since: 2.0

See Also: SimpleTagSupport

JSP.13.6.1.1 Methods

public void **doTag**()

Called by the container to invoke this tag. The implementation of this method is provided by the tag library developer, and handles all tag processing, body iteration, etc.

The JSP container will resynchronize any AT_BEGIN and AT_END variables (defined by the associated tag file, TagExtraInfo, or TLD) after the invocation of doTag().

Throws:
JspException - If an error occurred while processing this tag.

SkipPageException - If the page that (either directly or indirectly) invoked this tag is to cease evaluation. A Simple Tag Handler generated from a tag file must throw this exception if an invoked Classic Tag Handler returned SKIP_PAGE or if an invoked Simple Tag Handler threw SkipPageException or if an invoked Jsp Fragment threw a SkipPageException.

java.io.IOException - If there was an error writing to the output stream.

public JspTag **getParent**()

Returns the parent of this tag, for collaboration purposes.

Returns: the parent of this tag

public void **setJspBody**(JspFragment jspBody)

Provides the body of this tag as a JspFragment object, able to be invoked zero or more times by the tag handler.

This method is invoked by the JSP page implementation object prior to doTag(). If the action element is empty in the page, this method is not called at all.

Parameters:
jspBody - The fragment encapsulating the body of this tag.

public void **setJspContext**(JspContext pc)

Called by the container to provide this tag handler with the JspContext for this invocation. An implementation should save this value.

Parameters:
pc - the page context for this invocation

See Also: public void setPageContext(PageContext pc)

public void **setParent**(JspTag parent)

Sets the parent of this tag, for collaboration purposes.

The container invokes this method only if this tag invocation is nested within another tag invocation.

Parameters:
parent - the tag that encloses this tag

JSP.13.6.2 SimpleTagSupport

Syntax

public class SimpleTagSupport implements SimpleTag

All Implemented Interfaces: JspTag, SimpleTag

Description

A base class for defining tag handlers implementing SimpleTag.

The SimpleTagSupport class is a utility class intended to be used as the base class for new simple tag handlers. The SimpleTagSupport class implements the SimpleTag interface and adds additional convenience methods including getter methods for the properties in SimpleTag.

Since: 2.0

JSP.13.6.2.1 Constructors

public **SimpleTagSupport**()

Sole constructor. (For invocation by subclass constructors, typically implicit.)

JSP.13.6.2.2 Methods

public void **doTag**()

Default processing of the tag does nothing.

Throws:
JspException - Subclasses can throw JspException to indicate an error occurred while processing this tag.

SkipPageException - If the page that (either directly or indirectly) invoked this tag is to cease evaluation. A Simple Tag Handler generated from a tag file must throw this exception if an invoked Classic Tag Handler returned SKIP_PAGE or if an invoked Simple Tag Handler threw SkipPageException or if an invoked Jsp Fragment threw a SkipPageException.

IOException - Subclasses can throw IOException if there was an error writing to the output stream

See Also: public void doTag()

public static final JspTag **findAncestorWithClass**(JspTag from, java.lang.Class klass)

Find the instance of a given class type that is closest to a given instance. This method uses the getParent method from the Tag and/or SimpleTag interfaces. This method is used for coordination among cooperating tags.

For every instance of TagAdapter encountered while traversing the ancestors, the tag handler returned by TagAdapter.getAdaptee() - instead of the Tag-Adpater itself - is compared to klass. If the tag handler matches, it - and not its TagAdapter - is returned.

The current version of the specification only provides one formal way of indicating the observable type of a tag handler: its tag handler implementation class, described in the tag-class subelement of the tag element. This is extended in an informal manner by allowing the tag library author to indicate in the description subelement an observable type. The type should be a subtype of the tag handler implementation class or void. This addititional constraint can be exploited by a specialized container that knows about that specific tag library, as in the case of the JSP standard tag library.

When a tag library author provides information on the observable type of a tag handler, client programmatic code should adhere to that constraint. Specifically, the Class passed to findAncestorWithClass should be a subtype of the observable type.

Parameters:
from - The instance from where to start looking.

klass - The subclass of JspTag or interface to be matched

Returns: the nearest ancestor that implements the interface or is an instance of the class specified

protected JspFragment **getJspBody**()

Returns the body passed in by the container via setJspBody.

Returns: the fragment encapsulating the body of this tag, or null if the action element is empty in the page.

protected JspContext **getJspContext**()

Returns the page context passed in by the container via setJspContext.

Returns: the page context for this invocation

public JspTag **getParent**()

Returns the parent of this tag, for collaboration purposes.

Returns: the parent of this tag

public void **setJspBody**(JspFragment jspBody)

Stores the provided JspFragment.

Parameters:
jspBody - The fragment encapsulating the body of this tag. If the action element is empty in the page, this method is not called at all.

See Also: public void setJspBody(JspFragment jspBody)

public void **setJspContext**(JspContext pc)

Stores the provided JSP context in the private jspContext field. Subclasses can access the JspContext via getJspContext().

Parameters:
pc - the page context for this invocation

See Also: public void setJspContext(JspContext pc)

public void **setParent**(JspTag parent)

Sets the parent of this tag, for collaboration purposes.

The container invokes this method only if this tag invocation is nested within another tag invocation.

Parameters:
parent - the tag that encloses this tag

JSP.13.6.3 TagAdapter

Syntax

public class TagAdapter implements Tag

All Implemented Interfaces: JspTag, Tag

Description

Wraps any SimpleTag and exposes it using a Tag interface. This is used to allow collaboration between classic Tag handlers and SimpleTag handlers.

Because SimpleTag does not extend Tag, and because Tag.setParent() only accepts a Tag instance, a classic tag handler (one that implements Tag) cannot have a SimpleTag as its parent. To remedy this, a TagAdapter is created to wrap the SimpleTag parent, and the adapter is passed to setParent() instead. A classic Tag Handler can call getAdaptee() to retrieve the encapsulated SimpleTag instance.

Since: 2.0

JSP.13.6.3.1 Constructors

public **TagAdapter**(SimpleTag adaptee)

Creates a new TagAdapter that wraps the given SimpleTag and returns the parent tag when getParent() is called.

Parameters:
adaptee - The SimpleTag being adapted as a Tag.

JSP.13.6.3.2 Methods

public int **doEndTag**()

Must not be called.

Returns: always throws UnsupportedOperationException

Throws:
UnsupportedOperationException - Must not be called

JspException - never thrown

public int **doStartTag**()

Must not be called.

Returns: always throws UnsupportedOperationException

Throws:
UnsupportedOperationException - Must not be called

JspException - never thrown

public JspTag **getAdaptee**()

Gets the tag that is being adapted to the Tag interface. This should be an instance of SimpleTag in JSP 2.0, but room is left for other kinds of tags in future spec versions.

Returns: the tag that is being adapted

public Tag **getParent**()

Returns the parent of this tag, which is always getAdaptee().getParent(). This will either be the enclosing Tag (if getAdaptee().getParent() implements Tag), or an adapter to the enclosing Tag (if getAdaptee().getParent() does not implement Tag).

Returns: The parent of the tag being adapted.

public void **release**()

Must not be called.

Throws:
UnsupportedOperationException - Must not be called

public void **setPageContext**(PageContext pc)

Must not be called.

Parameters:
pc - ignored.

Throws:
UnsupportedOperationException - Must not be called

public void **setParent**(Tag parentTag)

Must not be called. The parent of this tag is always getAdaptee().getParent().

Parameters:
parentTag - ignored.

Throws:
UnsupportedOperationException - Must not be called.

JSP.13.7 JSP Fragments

JSP Fragments are represented in Java by an instance of the javax.servlet.jsp.tagext.JspFragment abstract class. Pieces of JSP code are translated into JSP fragments in the context of a tag invocation. JSP Fragments are created when providing the body of a <jsp:attribute> standard action for an attribute that is defined as a fragment or of type JspFragment, or when providing the body of a tag invocation handled by a Simple Tag Handler.

Before being passed to a tag handler, the JspFragment instance is associated with the JspContext of the surrounding page in an implementation-dependent manner. In addition, it is associated with the parent Tag or SimpleTag instance for collaboration purposes, so that when a custom action is invoked from within the fragment, setParent() can be called with the appropriate value. The fragment implementation must keep these associations for the duration of the tag invocation in which it is used.

The invoke() method executes the body and directs all output to either the passed in java.io.Writer or the JspWriter returned by the getOut() method of the JspContext associated with the fragment.

The implementation of each method can optionally throw a JspException, which must be handled by the invoker. Note that tag library developers and page authors should not generate JspFragment implementations manually.

The following sections specify the creation and invocation lifecycles of a JSP Fragment in detail, from the JSP Container's perspective.

Creation of a JSP Fragment

When a JSP fragment is created, the following steps occur (in order):

1. An instance of a class implementing the JspFragment abstract class is obtained (may either be created or can optionally be cached) each time the tag is invoked. This instance must be configured to produce the contents of the body of the fragment when invoked. If the fragment is defining the body of a <jsp:attribute>, the fragment must evaluate the body each time it is invoked. Otherwise, if the fragment is defining the body of a simple tag, the behavior of the fragment when invoked varies depending on the body-content declared for the tag:

   •If the body-content is "tagdependent", then the fragment must echo the contents of the body verbatim when invoked.

   •If the body-content is "scriptless", then the fragment must evaluate the body each time it is invoked.

2. The JspFragment instance is passed a reference to the current JspContext. Whenever the fragment invokes a tag handler, it must use this value when calling setJspContext().

3. The JspFragment instance is associated with an instance of the tag handler of the nearest enclosing tag invocation, or with null if there is no enclosing tag. Whenever the fragment invokes a tag handler, the fragment must use this value when calling setParent().

Invocation of a JSP Fragment

After a JSP fragment is created, it is passed to a tag handler for later invocation. JSP fragments can be invoked either programmatically from a tag handler written in Java, or from a tag file using the <jsp:invoke> or <jsp:doBody> standard action.

JSP fragments are passed to tag handlers using a bean property of type JspFragment. These fragments can be invoked by calling the invoke() method in the JspFragment abstract class. Note that it is legal (and possible) for a fragment to recursively invoke itself, indirectly.

The following steps are followed when invoking a JSP fragment:

1. The tag handler invoking the fragment is responsible for setting the values of all declared AT_BEGIN and NESTED variables in the JspContext of the calling page/tag, before invoking the fragment. Note that this is not always the same as the JspContext of the fragment being invoked, as fragments can be passed from one tag to another. In the case of tag files, for each variable declared in scope AT_BEGIN or NESTED, if a page scoped attribute exists with the provided name in the tag file, the JSP container must generate code to create/update the page scoped attribute of the provided name in the calling page/tag. If a page scoped attribute with the provided name does not exist in the calling page, and a page scoped attribute of the provided name is present in the tag file, the scoped attribute is removed from the tag file's page scope. See the chapter on Tag Files for details.
2. If <jsp:invoke> or <jsp:doBody> is being used to invoke a fragment, if the var attribute is specified, a custom java.io.Writer is created that can expose the result of the invocation as a java.lang.String object. If the varReader attribute is specified, a custom java.io.Writer object is created that can expose the resulting invocation as a java.io.Reader object.
3. The invoke() method of the fragment is invoked, passing in an optional Writer.
4. Before executing the body of the fragment, if a non-null value is provided for the writer parameter, then the value of JspContext.getOut() and the implicit "out" object must be updated to send output to that writer. To accomplish this, the container must call pushBody(writer) on the current JspContext, where writer is the instance of java.io.Writer passed to the fragment upon invocation.
5. The body of the fragment is then evaluated by executing the generated code. The body of the fragment may execute other standard or custom actions. If a classic Custom Tag Handler is invoked and returns SKIP_PAGE, or if a Simple Tag Handler is invoked and throws SkipPageException, the JspFragment must throw SkipPageException to signal that the calling page is to be skipped.
6. Once the fragment has completed its evaluation, even if an exception is thrown, the value of JspContext.getOut() must be restored via a call to popBody() on the current JspContext.

7. The fragment returns from invoke()
8. If <jsp:invoke> or <jsp:doBody> is being used to invoke a fragment, if the var or varReader attribute is specified, a scoped variable with a name equal to the value of the var or varReader attribute is created (or modified) in the page scope, and the value is set to a java.lang.String or java.io.Reader respectively that can produce the results of the fragment invocation.
9. The invoke() method can be called again, zero or more times. When the tag invocation defining the fragment is complete, the tag must discard the fragment instance since it might be reused by the container.

JSP.13.7.1 JspFragment

Syntax

public abstract class JspFragment

Description

Encapsulates a portion of JSP code in an object that can be invoked as many times as needed. JSP Fragments are defined using JSP syntax as the body of a tag for an invocation to a SimpleTag handler, or as the body of a <jsp:attribute> standard action specifying the value of an attribute that is declared as a fragment, or to be of type JspFragment in the TLD.

The definition of the JSP fragment must only contain template text and JSP action elements. In other words, it must not contain scriptlets or scriptlet expressions. At translation time, the container generates an implementation of the JspFragment abstract class capable of executing the defined fragment.

A tag handler can invoke the fragment zero or more times, or pass it along to other tags, before returning. To communicate values to/from a JSP fragment, tag handlers store/retrieve values in the JspContext associated with the fragment.

Note that tag library developers and page authors should not generate JspFragment implementations manually.

Implementation Note: It is not necessary to generate a separate class for each fragment. One possible implementation is to generate a single helper class for each page that implements JspFragment. Upon construction, a discriminator can be passed to select which fragment that instance will execute.

Since: 2.0

JSP.13.7.1.1 Constructors

public **JspFragment**()

JSP.13.7.1.2 Methods

public abstract JspContext **getJspContext**()

Returns the JspContext that is bound to this JspFragment.

Returns: The JspContext used by this fragment at invocation time.

public abstract void **invoke**(java.io.Writer out)

Executes the fragment and directs all output to the given Writer, or the JspWriter returned by the getOut() method of the JspContext associated with the fragment if out is null.

Parameters:
out - The Writer to output the fragment to, or null if output should be sent to JspContext.getOut().

Throws:
JspException - Thrown if an error occured while invoking this fragment.

SkipPageException - Thrown if the page that (either directly or indirectly) invoked the tag handler that invoked this fragment is to cease evaluation. The container must throw this exception if a Classic Tag Handler returned Tag.SKIP_PAGE or if a Simple Tag Handler threw SkipPageException.

java.io.IOException - If there was an error writing to the stream.

JSP.13.8 Example Simple Tag Handler Scenario

The following non-normative example is intended to help solidify some of the concepts relating to Tag Files, JSP Fragments and Simple Tag Handlers. In the first section, two sample input files are presented, a JSP (my.jsp), and a simple tag handler implemented using a tag file (simpletag.tag). One possible output of the translation process is presented in the second section.

Although short, the example shows all concepts, including the variable directive. In practice most uses of tags will be much simpler, but probably longer.

The sample generated code is annotated with comments that point to lifecycle steps presented in various sections. The notation is as follows:

- "Step T.x" = Annotated step x from "Lifecycle of Simple Tag Handlers" earlier in this Chapter.
- "Step C.x" = Annotated step x from "Creation of a JSP Fragment" earlier in this Chapter.

•“Step F.x” = Annotated step x from “Invocation of a JSP Fragment” earlier in this Chapter.

Sample Source Files

This section presents the sample source files in this scenario, from which the output files are generated.

Original JSP (my.jsp)

```
<%@ taglib prefix=“my” tagdir=“/WEB-INF/tags” %>
<my:simpleTag x=“10”>
   <jsp:attribute name=“y”>20</jsp:attribute>
   <jsp:attribute name=“nonfragment”>
      Nonfragment Template Text
   </jsp:attribute>
   <jsp:attribute name=“frag”>
      Fragment Template Text ${var1}
   </jsp:attribute>
   <jsp:body>
      Body of tag that defines an AT_BEGIN
      scripting variable ${var1}.
   </jsp:body>
</my:simpleTag>
```

Original Tag File (/WEB-INF/tags/simpletag.tag)

```
<%-- /WEB-INF/tags/simpletag.tag --%>
<%@ attribute name=“x” %>
<%@ attribute name=“y” %>
<%@ attribute name=“nonfragment” %>
<%@ attribute name=“frag” fragment=“true” %>
<%@ variable name-given=“var1” scope=“AT_BEGIN” %>
<%@ taglib prefix=“c” uri=“http://java.sun.com/jsp/jstl/core” %>
Some template text.
<c:set var=“var1” value=“${x+y}”/>
<jsp:invoke fragment=“frag” varReader=“var1”/>
Invoke the body:
<jsp:doBody/>
```

Sample Generated Files

This section presents sample output files that might be generated by a JSP compiler, from the source files presented in the previous section.

Helper class for JspFragment (JspFragmentBase.java)

```
public abstract class JspFragmentBase
   implements javax.servlet.jsp.tagext.JspFragment
{
   protected javax.servlet.jsp.JspContext jspContext;
   protected javax.servlet.jsp.tagext.JspTag parentTag;
   public void JspFragmentBase(
      javax.servlet.jsp.JspContext jspContext,
      javax.servlet.jsp.tagext.JspTag parentTag )
   {
      this.jspContext = jspContext;
      this.parentTag = parentTag;
   }
}
```

Relevant Portion of JSP Service Method

```
// Step T.1 - Initial creation
MySimpleTag _jsp_mySimpleTag = new MySimpleTag();
// Step T.2 - Set page context and parent (since parent is null,
// no need to call setParent() in this case)
_jsp_mySimpleTag.setJspContext( jspContext );
// Step T.3 - XML element attributes evaluated and set
_jsp.mySimpleTag.setX( “10” );
// Step T.4 - <jsp:attribute> elements evaluated and set
//   - parameter y
// (using PageContext.pushBody() is one possible implementation -
// one limitation is that this code will only work for Servlet-based code).
out = ((PageContext)jspContext).pushBody();
out.write( “20” );
_jsp_mySimpleTag.setY(
   ((javax.servlet.jsp.tagext.BodyContent)out).getString() );
out = jspContext.popBody();
//   - parameter nonfragment
// (using PageContext.pushBody() is one possible implementation -
// one limitation is that this code will only work for Servlet-based code).
// Note that trim is enabled by default, else we would have “\n    Non...”
out = ((PageContext)jspContext).pushBody();
out.write( “Nonfragment Template Text” );
_jsp_mySimpleTag.setNonfragment(
   ((javax.servlet.jsp.tagext.BodyContent)out).getString() );
out = jspContext.popBody();
//   - parameter frag
_jsp_mySimpleTag.setFrag(
   // Step C.1 - New instance of fragment created
   // Step C.2 - Store jspContext
   // Step C.3 - Association with nearest enclosing Tag instance
   new JspFragmentBase( jspContext, _jsp_mySimpleTag ) {
      public void invoke( java.io.Writer writer ) {
   javax.servlet.jsp.JspWriter out;
         // Step F.1-F.3 done in tag file (see following example)
   // Step F.4 - If writer provided, push body:
   if( out == null ) {
```

```
out = this.jspContext.getOut();
   }
   else {
out = this.jspContext.pushBody( writer );
   }
   // Step F.5 - Evaluate body of fragment:
         try {
            out.write( “Fragment Template Text ” );
            out.write( jspContext.getExpressionEvaluator().evaluate(
   “${var1}”,
   java.lang.String.class,
   vResolver, fMapper, “my” ) );
         }
         finally {
            // Step F.6 - Restore value of JspContext.getOut()
if( writer != null ) {
   this.jspContext.popBody();
}
         }
         // Step F.7-F.9 done in tag file (see following example)
      }
   } );
// Step T.5 - Determine and set body of the tag
// - body of tag
_jsp_mySimpleTag.setJspBody(
   // Step C.1 - New instance of fragment created
   // Step C.2 - Store jspContext
   // Step C.3 - Association with nearest enclosing Tag instance
   new JspFragmentBase( jspContext, _jsp_mySimpleTag ) {
      public void invoke( java.io.Writer writer ) {
   javax.servlet.jsp.JspWriter out;
         // Step F.1-F.3 done in tag file (see following example)
   // Step F.4 - If writer provided, push body:
   if( writer == null ) {
out = this.jspContext.getOut();
   }
   else {
out = this.jspContext.pushBody( writer );
   }
   // Step F.5 - Evaluate body of fragment:
         try {
            out.write(
               “Body of tag that defines an AT_BEGIN\n” +
               “ scripting variable ” );
            out.write( jspContext.getExpressionEvaluator().evaluate(
   “${var1}”,
   java.lang.String.class,
   vResolver, fMapper, “my” ) );
            out.write( “.\n” );
         }
         finally {
            // Step F.6 - Restore value of JspContext.getOut()
if( writer != null ) {
```

```
    this.jspContext.popBody();
}
          }
          // Step F.7-F.9 done in tag file (see following example)
       }
    } );
// Step T.6 - Inovke doTag
// Step T.7 occurs in the tag file (see following example)
// Step T.8 - doTag returns - page will catch SkipPageException.
_jsp_mySimpleTag.doTag();
// Step T.9 - Declare AT_BEGIN and AT_END scripting variables
String var1 = (String)jspContext.findAttribute( “var1” );
```

Generated Simple Tag Handler (MySimpleTag.java)

```
public class MySimpleTag
    extends javax.servlet.jsp.tagext.SimpleTagSupport
{
    // Attributes:
    private String x;
    private String y;
    private String nonfragment;
    private javax.servlet.jsp.tagext.JspFragment frag;
    // Setters and getters for attributes:
    public void setX( Stirng x ) {
        this.x = x;
    }
    public String getX() {
        return this.x;
    }
    public void setY( String y ) {
        this.y = y;
    }
    public String getY() {
        return this.y;
    }
    public void setNonfragment( String nonfragment ) {
        this.nonfragment = nonfragment;
    }
    public String getNonfragment() {
        return this.nonfragment;
    }
    public void setFrag( javax.servlet.jsp.tagext.JspFragment frag ) {
        this.frag = frag;
    }
    public javax.servlet.jsp.tagext.JspFragment getFrag() {
        return this.frag;
    }
    protected JspContext jspContext;
    public void setJspContext( JspContext ctx ) {
super.setJspContext( ctx );
// Step T.2 - A JspContext wrapper is created.
// (Implementation of wrapper not shown).
```

```
this.jspContext = new utils.JspContextWrapper( ctx );
    }
    public JspContext getJspContext() {
// Step T.2 - Calling getJspContext() must return the
// wrapped JspContext.
return this.jspContext;
    }
    public void doTag() throws JspException {
java.lang.Object jspValue;
JspContext jspContext = getJspContext();
JspContext _jsp_parentContext =
    SimpleTagSupport.this.getJspContext();
try {
    javax.servlet.jsp.JspWriter out = jspContext.getOut();
    // Create page-scope attributes for each tag attribute:
    this.jspContext.setAttribute( “x”, getX() );
    this.jspContext.setAttribute( “y”, getY() );
    this.jspContext.setAttribute( “nonfragment”, getNonfragment() );
    this.jspContext.setAttribute( “frag”, getFrag() );
    // Synchronize AT_BEGIN variables from calling page
    if( (jspValue = _jsp_parentContext.getAttribute(
    “var1” )) != null )
    {
jspContext.setAttribute( “var1”, value );
    }
    else {
jspContext.removeAttribute( “var1”,
    JspContext.PAGE_SCOPE );
    }
    // Tag template text:
    out.write( “\n\n\n\n\n\n\n\nSome template text.\n” );
    // Invoke c:set - recognized tag handler from JSTL:
    jspContext.setAttribute( “var1”,
jspContext.getExpressionEvaluator().evaluate(
    “${x+y}”,
    java.lang.String.class,
    jspContext,
    prefixMap, functionMap, “my” ) );
    // Invoke the “frag” fragment:
    // Step F.1 - Set values of AT_BEGIN and NESTED variables
    //     in calling page context.
    if( (jspValue = jspContext.getAttribute( “var1” )) != null ) {
_jsp_parentContext.setAttribute( “var1”, value );
    }
    else {
_jsp_parentContext.removeAttribute( “var1”,
    JspContext.PAGE_SCOPE );
    }
    // Step F.2 - varReader is specified, generate a writer.
    java.io.Writer _jsp_sout = new java.io.StringWriter();
    // Step F.3 - Invoke fragment with writer
    getFrag().invoke( _jsp_sout );
    // Step F.4 - F.6 occur in the fragment (see above)
```

```
    // Step F.7 - fragment returns
    // Step F.8 - varReader specified, so save to var
    jspContext.setAttribute(
“var1”, new StringReader( _jsp_sout.toString() ) );
    // Step F.9 - Done!
    out.write( “\n\nInvoke the body:\n” );
    // Invoke the body of the tag:
    // Step F.1 - Set values of AT_BEGIN and NESTED variables
    //     in calling page context.
    if( (jspValue = jspContext.getAttribute( “var1” )) != null ) {
_jsp_parentContext.setAttribute( “var1”, value );
    }
    else {
_jsp_parentContext.removeAttribute( “var1”,
    JspContext.PAGE_SCOPE);
    }
    // Step F.2 - varReader is not specified - does not apply.
    try {
// Step F.3 - Invoke body, passing optional writer
getJspBody().invoke( null );
    }
    finally {
// Steps F.4 - F.6 occur in the fragment (see above)
// Step F.7 - fragment returns
    }
    // Step F.8 does not apply.
    // Step F.9 - Done!
}
finally {
    // Tag handlers generate code to synchronize AT_BEGIN with
    // calling page, regardless of whether an error occurs.
    if( (jspValue = jspContext.getAttribute( “var1” )) != null ) {
_jsp_parentContext.setAttribute( “var1”, value );
    }
    else {
_jsp_parentContext.removeAttribute( “var1”,
    JspContext.PAGE_SCOPE );
    }
}
    }
}
```

JSP.13.9 Translation-time Classes

The next classes are used at translation time.

Tag mapping, Tag name

A taglib directive introduces a tag library and associates a prefix to it. The TLD associated with the library associates Tag handler classes (plus other information)

with tag names. This information is used to associate a Tag class, a prefix, and a name with each custom action element appearing in a JSP page.

At execution time the implementation of a JSP page will use an available Tag instance with the appropriate property settings and then follow the protocol described by the interfaces Tag, IterationTag, BodyTag, SimpleTag, and TryCatchFinally. The implementation guarantees that all tag handler instances are initialized and all are released, but the implementation can assume that previous settings are preserved by a tag handler, to reduce run-time costs.

Scripting Variables

JSP supports scripting variables that can be declared within a scriptlet and can be used in another. JSP actions also can be used to define scripting variables so they can used in scripting elements, or in other actions. This is very useful in some cases; for example, the jsp:useBean standard action may define an object which can later be used through a scripting variable.

In some cases the information on scripting variables can be described directly into the TLD using elements. A special case is typical interpretation of the “id” attribute. In other cases the logic that decides whether an action instance will define a scripting variable may be quite complex and the name of a TagExtraInfo class is instead given in the TLD. The getVariableInfo method of this class is used at translation time to obtain information on each variable that will be created at request time when this action is executed. The method is passed a TagData instance that contains the translation-time attribute values.

Validation

The TLD file contains several pieces of information that is used to do syntactic validation at translation-time. It also contains two extensible validation mechanisms: a TagLibraryValidator class can be used to validate a complete JSP page, and a TagExtraInfo class can be used to validate a specific action. In some cases, additional request-time validation will be done dynamically within the methods in the Tag instance. If an error is discovered, an instance of JspTagException can be thrown. If uncaught, this object will invoke the errorpage mechanism of JSP.

The TagLibraryValidator is an addition to the JSP 1.2 specification and is very open ended, being strictly more powerful than the TagExtraInfo mechanism. A JSP page is presented via the PageData object, which abstracts the XML view of the JSP page.

A PageData instance will provides an InputStream (read-only) on the page. Later specifications may add other views on the page (DOM, SAX, JDOM are all candidates), for now these views can be generated from the InputStream and

perhaps can be cached for improved performance (recall the view of the page is just read-only).

As of JSP 2.0, the JSP container must support a jsp:id attribute to provide higher quality validation errors. The container will track the JSP pages as passed to the container, and will assign to each element a unique "id", which is passed as the value of the jsp:id attribute. Each XML element in the XML view will be extended with this attribute. The TagLibraryValidator can use the attribute in one or more ValidationMessage objects. The container then, in turn, can use these values to provide more precise information on the location of an error.

The prefix for the id attribute need not be "jsp" but it must map to the namespace http://java.sun.com/JSP/Page. In the case where the user has redefined the jsp prefix, an alternative prefix must be used by the container.

Validation Details

In detail, validation is done as follows:

First, the JSP page is parsed using the information in the TLD. At this stage valid mandatory and optional attributes are checked.

Second, for each unique tag library in the page as determined by the tag library URI, and in the lexical order in which they appear, their associated validator class (if any) is invoked. This involves several substeps.

The first substep is to obtain an initialized validator instance by either:

- construct a new instance and invoke setInitParameters() on it, or
- obtain an existing instance that is not being used, invoke release() on it, and then invoke setInitParameters() on it, or
- locate an existing instance that is not being used on which the desired setInitParameters() has already been invoked

The class name is as indicated in the <validator-class> element, and the Map passed through setInitParameters() is as described in the <init-params> element. All TagLibraryValidator classes are supposed to keep their initParameters until new ones are set, or until release() is invoked on them.

The second substep is to perform the actual validation. This is done by invoking the validate() method with a prefix, uri, and PageData that correspond to the taglib directive instance being validated and the PageData representing the page. In the case where a single URI is mapped to more than one prefix, the prefix of the first URI must be used.

The last substep is to invoke the release() method on the validator tag when it is no longer needed. This method releases all resources.

Finally, after checking all the tag library validator classes, the TagExtraInfo classes for all tags will be consulted by invoking their validate method. The order of invocation of this methods is undefined.

JSP.13.9.1 TagLibraryInfo

Syntax

public abstract class TagLibraryInfo

Description

Translation-time information associated with a taglib directive, and its underlying TLD file. Most of the information is directly from the TLD, except for the prefix and the uri values used in the taglib directive

JSP.13.9.1.1 Fields

protected FunctionInfo[] **functions**

An array describing the functions that are defined in this tag library.

Since: 2.0

protected java.lang.String **info**

Information (documentation) for this TLD.

protected java.lang.String **jspversion**

The version of the JSP specification this tag library is written to.

protected java.lang.String **prefix**

The prefix assigned to this taglib from the taglib directive.

protected java.lang.String **shortname**

The preferred short name (prefix) as indicated in the TLD.

protected TagFileInfo[] **tagFiles**

An array describing the tag files that are defined in this tag library.

Since: 2.0

protected TagInfo[] **tags**

An array describing the tags that are defined in this tag library.

protected java.lang.String **tlibversion**

The version of the tag library.

protected java.lang.String **uri**

The value of the uri attribute from the taglib directive for this library.

protected java.lang.String **urn**

The “reliable” URN indicated in the TLD.

JSP.13.9.1.2 Constructors

protected **TagLibraryInfo**(java.lang.String prefix, java.lang.String uri)

Constructor. This will invoke the constructors for TagInfo, and TagAttributeInfo after parsing the TLD file.

Parameters:
prefix - the prefix actually used by the taglib directive

uri - the URI actually used by the taglib directive

JSP.13.9.1.3 Methods

public FunctionInfo **getFunction**(java.lang.String name)

Get the FunctionInfo for a given function name, looking through all the functions in this tag library.

Parameters:
name - The name (no prefix) of the function

Returns: the FunctionInfo for the function with the given name, or null if no such function exists

Since: 2.0

public FunctionInfo[] **getFunctions**()

An array describing the functions that are defined in this tag library.

Returns: the functions defined in this tag library, or a zero length array if the tag library defines no functions.

Since: 2.0

public java.lang.String **getInfoString**()

Information (documentation) for this TLD.

Returns: the info string for this tag lib

public java.lang.String **getPrefixString**()

The prefix assigned to this taglib from the taglib directive

Returns: the prefix assigned to this taglib from the taglib directive

public java.lang.String **getReliableURN**()

The “reliable” URN indicated in the TLD (the uri element). This may be used by authoring tools as a global identifier to use when creating a taglib directive for this library.

Returns: a reliable URN to a TLD like this

public java.lang.String **getRequiredVersion**()

A string describing the required version of the JSP container.

Returns: the (minimal) required version of the JSP container.

See Also: JspEngineInfo

public java.lang.String **getShortName**()

The preferred short name (prefix) as indicated in the TLD. This may be used by authoring tools as the preferred prefix to use when creating an taglib directive for this library.

Returns: the preferred short name for the library

public TagInfo **getTag**(java.lang.String shortname)

Get the TagInfo for a given tag name, looking through all the tags in this tag library.

Parameters:
shortname - The short name (no prefix) of the tag

Returns: the TagInfo for the tag with the specified short name, or null if no such tag is found

public TagFileInfo **getTagFile**(java.lang.String shortname)

Get the TagFileInfo for a given tag name, looking through all the tag files in this tag library.

Parameters:
shortname - The short name (no prefix) of the tag

Returns: the TagFileInfo for the specified Tag file, or null if no Tag file is found

Since: 2.0

public TagFileInfo[] **getTagFiles**()

An array describing the tag files that are defined in this tag library.

Returns: the TagFileInfo objects corresponding to the tag files defined by this tag library, or a zero length array if this tag library defines no tags files

Since: 2.0

public TagInfo[] **getTags**()

An array describing the tags that are defined in this tag library.

Returns: the TagInfo objects corresponding to the tags defined by this tag library, or a zero length array if this tag library defines no tags

public java.lang.String **getURI**()

The value of the uri attribute from the taglib directive for this library.

Returns: the value of the uri attribute

JSP.13.9.2 TagInfo

Syntax

public class TagInfo

Description

Tag information for a tag in a Tag Library; This class is instantiated from the Tag Library Descriptor file (TLD) and is available only at translation time.

JSP.13.9.2.1 Fields

public static final java.lang.String **BODY_CONTENT_EMPTY**

Static constant for getBodyContent() when it is empty.

public static final java.lang.String **BODY_CONTENT_JSP**

Static constant for getBodyContent() when it is JSP.

public static final java.lang.String **BODY_CONTENT_SCRIPTLESS**

Static constant for getBodyContent() when it is scriptless.

Since: 2.0

public static final java.lang.String **BODY_CONTENT_TAG_DEPENDENT**

Static constant for getBodyContent() when it is Tag dependent.

JSP.13.9.2.2 Constructors

public **TagInfo**(java.lang.String tagName, java.lang.String tagClassName,
java.lang.String bodycontent, java.lang.String infoString,
TagLibraryInfo taglib, TagExtraInfo tagExtraInfo,
TagAttributeInfo[] attributeInfo)

Constructor for TagInfo from data in the JSP 1.1 format for TLD. This class is to be instantiated only from the TagLibrary code under request from some JSP code that is parsing a TLD (Tag Library Descriptor). Note that, since TagLibibraryInfo reflects both TLD information and taglib directive information, a TagInfo instance is dependent on a taglib directive. This is probably a design error, which may be fixed in the future.

Parameters:
tagName - The name of this tag

tagClassName - The name of the tag handler class

bodycontent - Information on the body content of these tags

infoString - The (optional) string information for this tag

taglib - The instance of the tag library that contains us.

tagExtraInfo - The instance providing extra Tag info. May be null

attributeInfo - An array of AttributeInfo data from descriptor. May be null;

public **TagInfo**(java.lang.String tagName, java.lang.String tagClassName, java.lang.String bodycontent, java.lang.String infoString, TagLibraryInfo taglib, TagExtraInfo tagExtraInfo, TagAttributeInfo[] attributeInfo, java.lang.String displayName, java.lang.String smallIcon, java.lang.String largeIcon, TagVariableInfo[] tvi)

Constructor for TagInfo from data in the JSP 1.2 format for TLD. This class is to be instantiated only from the TagLibrary code under request from some JSP code that is parsing a TLD (Tag Library Descriptor). Note that, since TagLibibraryInfo reflects both TLD information and taglib directive information, a TagInfo instance is dependent on a taglib directive. This is probably a design error, which may be fixed in the future.

Parameters:
tagName - The name of this tag

tagClassName - The name of the tag handler class

bodycontent - Information on the body content of these tags

infoString - The (optional) string information for this tag

taglib - The instance of the tag library that contains us.

tagExtraInfo - The instance providing extra Tag info. May be null

attributeInfo - An array of AttributeInfo data from descriptor. May be null;

displayName - A short name to be displayed by tools

smallIcon - Path to a small icon to be displayed by tools

largeIcon - Path to a large icon to be displayed by tools

tvi - An array of a TagVariableInfo (or null)

public **TagInfo**(java.lang.String tagName, java.lang.String tagClassName, java.lang.String bodycontent, java.lang.String infoString, TagLibraryInfo taglib, TagExtraInfo tagExtraInfo, TagAttributeInfo[] attributeInfo, java.lang.String displayName, java.lang.String smallIcon, java.lang.String largeIcon, TagVariableInfo[] tvi, boolean dynamicAttributes)

Constructor for TagInfo from data in the JSP 2.0 format for TLD. This class is to be instantiated only from the TagLibrary code under request from some

JSP code that is parsing a TLD (Tag Library Descriptor). Note that, since TagLibibraryInfo reflects both TLD information and taglib directive information, a TagInfo instance is dependent on a taglib directive. This is probably a design error, which may be fixed in the future.

Parameters:
tagName - The name of this tag

tagClassName - The name of the tag handler class

bodycontent - Information on the body content of these tags

infoString - The (optional) string information for this tag

taglib - The instance of the tag library that contains us.

tagExtraInfo - The instance providing extra Tag info. May be null

attributeInfo - An array of AttributeInfo data from descriptor. May be null;

displayName - A short name to be displayed by tools

smallIcon - Path to a small icon to be displayed by tools

largeIcon - Path to a large icon to be displayed by tools

tvi - An array of a TagVariableInfo (or null)

dynamicAttributes - True if supports dynamic attributes

Since: 2.0

JSP.13.9.2.3 Methods

public TagAttributeInfo[] **getAttributes**()

Attribute information (in the TLD) on this tag. The return is an array describing the attributes of this tag, as indicated in the TLD.

Returns: The array of TagAttributeInfo for this tag, or a zero-length array if the tag has no attributes.

public java.lang.String **getBodyContent**()

The bodycontent information for this tag. If the bodycontent is not defined for this tag, the default of JSP will be returned.

Returns: the body content string.

public java.lang.String **getDisplayName**()

Get the displayName.

Returns: A short name to be displayed by tools, or null if not defined

public java.lang.String **getInfoString**()

The information string for the tag.

Returns: the info string, or null if not defined

public java.lang.String **getLargeIcon**()

Get the path to the large icon.

Returns: Path to a large icon to be displayed by tools, or null if not defined

public java.lang.String **getSmallIcon**()

Get the path to the small icon.

Returns: Path to a small icon to be displayed by tools, or null if not defined

public java.lang.String **getTagClassName**()

Name of the class that provides the handler for this tag.

Returns: The name of the tag handler class.

public TagExtraInfo **getTagExtraInfo**()

The instance (if any) for extra tag information.

Returns: The TagExtraInfo instance, if any.

public TagLibraryInfo **getTagLibrary**()

The instance of TabLibraryInfo we belong to.

Returns: the tag library instance we belong to

public java.lang.String **getTagName**()

The name of the Tag.

Returns: The (short) name of the tag.

public TagVariableInfo[] **getTagVariableInfos**()

Get TagVariableInfo objects associated with this TagInfo.

Returns: Array of TagVariableInfo objects corresponding to variables declared by this tag, or a zero length array if no variables have been declared

public VariableInfo[] **getVariableInfo**(TagData data)

Information on the scripting objects created by this tag at runtime. This is a convenience method on the associated TagExtraInfo class.

Parameters:
data - TagData describing this action.

Returns: if a TagExtraInfo object is associated with this TagInfo, the result of getTagExtraInfo().getVariableInfo(data), otherwise null.

public boolean **hasDynamicAttributes**()

Get dynamicAttributes associated with this TagInfo.

Returns: True if tag handler supports dynamic attributes

Since: 2.0

public boolean **isValid**(TagData data)

Translation-time validation of the attributes. This is a convenience method on the associated TagExtraInfo class.

Parameters:
data - The translation-time TagData instance.

Returns: Whether the data is valid.

public void **setTagExtraInfo**(TagExtraInfo tei)

Set the instance for extra tag information.

Parameters:
tei - the TagExtraInfo instance

public void **setTagLibrary**(TagLibraryInfo tl)

Set the TagLibraryInfo property. Note that a TagLibraryInfo element is dependent not just on the TLD information but also on the specific taglib instance used. This means that a fair amount of work needs to be done to construct and initialize TagLib objects. If used carefully, this setter can be used to avoid having to create new TagInfo elements for each taglib directive.

Parameters:
tl - the TagLibraryInfo to assign

public ValidationMessage[] **validate**(TagData data)

Translation-time validation of the attributes. This is a convenience method on the associated TagExtraInfo class.

Parameters:
data - The translation-time TagData instance.

Returns: A null object, or zero length array if no errors, an array of ValidationMessages otherwise.

Since: 2.0

JSP.13.9.3 TagFileInfo

Syntax

public class TagFileInfo

Description

Tag information for a tag file in a Tag Library; This class is instantiated from the Tag Library Descriptor file (TLD) and is available only at translation time.

Since: 2.0

JSP.13.9.3.1 Constructors

public **TagFileInfo**(java.lang.String name, java.lang.String path, TagInfo tagInfo)

Constructor for TagFileInfo from data in the JSP 2.0 format for TLD. This class is to be instantiated only from the TagLibrary code under request from some JSP code that is parsing a TLD (Tag Library Descriptor). Note that, since TagLibibraryInfo reflects both TLD information and taglib directive information, a TagFileInfo instance is dependent on a taglib directive. This is probably a design error, which may be fixed in the future.

Parameters:
name - The unique action name of this tag

path - Where to find the .tag file implementing this action, relative to the location of the TLD file.

tagInfo - The detailed information about this tag, as parsed from the directives in the tag file.

JSP.13.9.3.2 Methods

public java.lang.String **getName**()

The unique action name of this tag.

Returns: The (short) name of the tag.

public java.lang.String **getPath**()

Where to find the .tag file implementing this action.

Returns: The path of the tag file, relative to the TLD, or "." if the tag file was defined in an implicit tag file.

public TagInfo **getTagInfo**()

Returns information about this tag, parsed from the directives in the tag file.

Returns: a TagInfo object containing information about this tag

JSP.13.9.4 TagAttributeInfo

Syntax

public class TagAttributeInfo

Description

Information on the attributes of a Tag, available at translation time. This class is instantiated from the Tag Library Descriptor file (TLD).

Only the information needed to generate code is included here. Other information like SCHEMA for validation belongs elsewhere.

JSP.13.9.4.1 Fields

public static final java.lang.String **ID**

"id" is wired in to be ID. There is no real benefit in having it be something else IDREFs are not handled any differently.

JSP.13.9.4.2 Constructors

public **TagAttributeInfo**(java.lang.String name, boolean required, java.lang.String type, boolean reqTime)

Constructor for TagAttributeInfo. This class is to be instantiated only from the TagLibrary code under request from some JSP code that is parsing a TLD (Tag Library Descriptor).

Parameters:
name - The name of the attribute.

required - If this attribute is required in tag instances.

type - The name of the type of the attribute.

reqTime - Whether this attribute holds a request-time Attribute.

public **TagAttributeInfo**(java.lang.String name, boolean required, java.lang.String type, boolean reqTime, boolean fragment)

JSP 2.0 Constructor for TagAttributeInfo. This class is to be instantiated only from the TagLibrary code under request from some JSP code that is parsing a TLD (Tag Library Descriptor).

Parameters:
name - The name of the attribute.

required - If this attribute is required in tag instances.

type - The name of the type of the attribute.

reqTime - Whether this attribute holds a request-time Attribute.

fragment - Whether this attribute is of type JspFragment

Since: 2.0

JSP.13.9.4.3 Methods

public boolean **canBeRequestTime**()

Whether this attribute can hold a request-time value.

Returns: if the attribute can hold a request-time value.

public static TagAttributeInfo **getIdAttribute**(TagAttributeInfo[] a)

Convenience static method that goes through an array of TagAttributeInfo objects and looks for “id”.

Parameters:
a - An array of TagAttributeInfo

Returns: The TagAttributeInfo reference with name “id”

public java.lang.String **getName**()

The name of this attribute.

Returns: the name of the attribute

public java.lang.String **getTypeName**()

The type (as a String) of this attribute.

Returns: the type of the attribute

public boolean **isFragment**()

Whether this attribute is of type JspFragment.

Returns: if the attribute is of type JspFragment

Since: 2.0

public boolean **isRequired**()

Whether this attribute is required.

Returns: if the attribute is required.

public java.lang.String **toString**()

Returns a String representation of this TagAttributeInfo, suitable for debugging purposes.

Overrides: java.lang.Object.toString() in class java.lang.Object

Returns: a String representation of this TagAttributeInfo

JSP.13.9.5 PageData

Syntax

public abstract class PageData

Description

Translation-time information on a JSP page. The information corresponds to the XML view of the JSP page.

Objects of this type are generated by the JSP translator, e.g. when being pased to a TagLibraryValidator instance.

JSP.13.9.5.1 Constructors

public **PageData**()

Sole constructor. (For invocation by subclass constructors, typically implicit.)

JSP.13.9.5.2 Methods

public abstract java.io.InputStream **getInputStream**()

Returns an input stream on the XML view of a JSP page. The stream is encoded in UTF-8. Recall tht the XML view of a JSP page has the include directives expanded.

Returns: An input stream on the document.

JSP.13.9.6 TagLibraryValidator

Syntax

public abstract class TagLibraryValidator

Description

Translation-time validator class for a JSP page. A validator operates on the XML view associated with the JSP page.

The TLD file associates a TagLibraryValidator class and some init arguments with a tag library.

The JSP container is reponsible for locating an appropriate instance of the appropriate subclass by

- new a fresh instance, or reuse an available one
- invoke the setInitParams(Map) method on the instance

once initialized, the validate(String, String, PageData) method will be invoked, where the first two arguments are the prefix and uri for this tag library in the XML View. The prefix is intended to make it easier to produce an error message. However, it is not always accurate. In the case where a single URI is mapped to more than one prefix in the XML view, the prefix of the first URI is provided. Therefore, to provide high quality error messages in cases where the tag elements themselves are checked, the prefix parameter should be ignored and the actual prefix of the element should be used instead. TagLibraryValidators should always use the uri to identify elements as beloning to the tag library, not the prefix.

A TagLibraryValidator instance may create auxiliary objects internally to perform the validation (e.g. an XSchema validator) and may reuse it for all the pages in a given translation run.

The JSP container is not guaranteed to serialize invocations of validate() method, and TagLibraryValidators should perform any synchronization they may require.

As of JSP 2.0, a JSP container must provide a jsp:id attribute to provide higher quality validation errors. The container will track the JSP pages as passed to the container, and will assign to each element a unique “id”, which is passed as the value of the jsp:id attribute. Each XML element in the XML view available will be extended with this attribute. The TagLibraryValidator can then use the attribute in one or more ValidationMessage objects. The container then, in turn, can use these values to provide more precise information on the location of an error.

The actual prefix of the id attribute may or may not be jsp but it will always map to the namespace http://java.sun.com/JSP/Page. A TagLibraryValidator implementation must rely on the uri, not the prefix, of the id attribute.

JSP.13.9.6.1 Constructors

public **TagLibraryValidator**()

Sole constructor. (For invocation by subclass constructors, typically implicit.)

JSP.13.9.6.2 Methods

public java.util.Map **getInitParameters**()

Get the init parameters data as an immutable Map. Parameter names are keys, and parameter values are the values.

Returns: The init parameters as an immutable map.

public void **release**()

Release any data kept by this instance for validation purposes.

public void **setInitParameters**(java.util.Map map)

Set the init data in the TLD for this validator. Parameter names are keys, and parameter values are the values.

Parameters:
map - A Map describing the init parameters

public ValidationMessage[] **validate**(java.lang.String prefix, java.lang.String uri, PageData page)

Validate a JSP page. This will get invoked once per unique tag library URI in the XML view. This method will return null if the page is valid; otherwise the method should return an array of ValidationMessage objects. An array of length zero is also interpreted as no errors.

Parameters:
prefix - the first prefix with which the tag library is associated, in the XML view. Note that some tags may use a different prefix if the namespace is redefined.

uri - the tag library's unique identifier

page - the JspData page object

Returns: A null object, or zero length array if no errors, an array of ValidationMessages otherwise.

JSP.13.9.7 ValidationMessage

Syntax

public class ValidationMessage

Description

A validation message from either TagLibraryValidator or TagExtraInfo.

As of JSP 2.0, a JSP container must support a jsp:id attribute to provide higher quality validation errors. The container will track the JSP pages as passed to the container, and will assign to each element a unique "id", which is passed as the value of the jsp:id attribute. Each XML element in the XML view available will be extended with this attribute. The TagLibraryValidator can then use the attribute in one or more ValidationMessage objects. The container then, in turn, can use these values to provide more precise information on the location of an error.

The actual prefix of the id attribute may or may not be jsp but it will always map to the namespace http://java.sun.com/JSP/Page. A TagLibraryValidator implementation must rely on the uri, not the prefix, of the id attribute.

JSP.13.9.7.1 Constructors

public **ValidationMessage**(java.lang.String id, java.lang.String message)

Create a ValidationMessage. The message String should be non-null. The value of id may be null, if the message is not specific to any XML element, or if no jsp:id attributes were passed on. If non-null, the value of id must be the value of a jsp:id attribute for the PageData passed into the validate() method.

Parameters:
id - Either null, or the value of a jsp:id attribute.

message - A localized validation message.

JSP.13.9.7.2 Methods

public java.lang.String **getId**()

Get the jsp:id. Null means that there is no information available.

Returns: The jsp:id information.

public java.lang.String **getMessage**()

Get the localized validation message.

Returns: A validation message

JSP.13.9.8 TagExtraInfo

Syntax

public abstract class TagExtraInfo

Description

Optional class provided by the tag library author to describe additional translation-time information not described in the TLD. The TagExtraInfo class is mentioned in the Tag Library Descriptor file (TLD).

This class can be used:

- to indicate that the tag defines scripting variables
- to perform translation-time validation of the tag attributes.

It is the responsibility of the JSP translator that the initial value to be returned by calls to getTagInfo() corresponds to a TagInfo object for the tag being translated. If an explicit call to setTagInfo() is done, then the object passed will be returned in subsequent calls to getTagInfo().

The only way to affect the value returned by getTagInfo() is through a setTagInfo() call, and thus, TagExtraInfo.setTagInfo() is to be called by the JSP translator, with a TagInfo object that corresponds to the tag being translated. The call should happen before any invocation on validate() and before any invocation on getVariableInfo().

NOTE: It is a (translation time) error for a tag definition in a TLD with one or more variable subelements to have an associated TagExtraInfo implementation that returns a VariableInfo array with one or more elements from a call to getVariableInfo().

JSP.13.9.8.1 Constructors

public **TagExtraInfo**()

Sole constructor. (For invocation by subclass constructors, typically implicit.)

JSP.13.9.8.2 Methods

public final TagInfo **getTagInfo**()

Get the TagInfo for this class.

Returns: the taginfo instance this instance is extending

public VariableInfo[] **getVariableInfo**(TagData data)

information on scripting variables defined by the tag associated with this TagExtraInfo instance. Request-time attributes are indicated as such in the TagData parameter.

Parameters:
data - The TagData instance.

Returns: An array of VariableInfo data, or null or a zero length array if no scripting variables are to be defined.

public boolean **isValid**(TagData data)

Translation-time validation of the attributes. Request-time attributes are indicated as such in the TagData parameter. Note that the preferred way to do validation is with the validate() method, since it can return more detailed information.

Parameters:
data - The TagData instance.

Returns: Whether this tag instance is valid.

See Also: public ValidationMessage[] validate(TagData data)

public final void **setTagInfo**(TagInfo tagInfo)

Set the TagInfo for this class.

Parameters:
tagInfo - The TagInfo this instance is extending

public ValidationMessage[] **validate**(TagData data)

Translation-time validation of the attributes. Request-time attributes are indicated as such in the TagData parameter. Because of the higher quality validation messages possible, this is the preferred way to do validation (although isValid() still works).

JSP 2.0 and higher containers call validate() instead of isValid(). The default implementation of this method is to call isValid(). If isValid() returns false, a generic ValidationMessage[] is returned indicating isValid() returned false.

Parameters:
data - The TagData instance.

Returns: A null object, or zero length array if no errors, an array of ValidationMessages otherwise.

Since: 2.0

JSP.13.9.9 TagData

Syntax

public class TagData implements java.lang.Cloneable

All Implemented Interfaces: java.lang.Cloneable

Description

The (translation-time only) attribute/value information for a tag instance.

TagData is only used as an argument to the isValid, validate, and getVariableInfo methods of TagExtraInfo, which are invoked at translation time.

JSP.13.9.9.1 Fields

public static final java.lang.Object **REQUEST_TIME_VALUE**

Distinguished value for an attribute to indicate its value is a request-time expression (which is not yet available because TagData instances are used at translation-time).

JSP.13.9.9.2 Constructors

public **TagData**(java.util.Hashtable attrs)

Constructor for a TagData. If you already have the attributes in a hashtable, use this constructor.

Parameters:
attrs - A hashtable to get the values from.

public **TagData**(java.lang.Object[][] atts)

Constructor for TagData.

A typical constructor may be

```
static final Object[][] att = {{“connection”, “conn0”},
{“id”, “query0”}};
static final TagData td = new TagData(att);
```

All values must be Strings except for those holding the distinguished object REQUEST_TIME_VALUE.

Parameters:
atts - the static attribute and values. May be null.

JSP.13.9.9.3 Methods

public java.lang.Object **getAttribute**(java.lang.String attName)

The value of the attribute. If a static value is specified for an attribute that accepts a request-time attribute expression then that static value is returned, even if the value is provided in the body of a action. The distinguished object REQUEST_TIME_VALUE is only returned if the value is specified as a request-time attribute expression or via the <jsp:attribute> action with a body that contains dynamic content (scriptlets, scripting expressions, EL expressions, standard actions, or custom actions). Returns null if the attribute is not set.

Parameters:
attName - the name of the attribute

Returns: the attribute’s value

public java.util.Enumeration **getAttributes**()

Enumerates the attributes.

Returns: An enumeration of the attributes in a TagData

public java.lang.String **getAttributeString**(java.lang.String attName)

Get the value for a given attribute.

Parameters:

attName - the name of the attribute

Returns: the attribute value string

Throws:
ClassCastException - if attribute value is not a String

public java.lang.String **getId**()

The value of the tag's id attribute.

Returns: the value of the tag's id attribute, or null if no such attribute was specified.

public void **setAttribute**(java.lang.String attName, java.lang.Object value)

Set the value of an attribute.

Parameters:
attName - the name of the attribute

value - the value.

JSP.13.9.10 VariableInfo

Syntax

public class VariableInfo

Description

Information on the scripting variables that are created/modified by a tag (at run-time). This information is provided by TagExtraInfo classes and it is used by the translation phase of JSP.

Scripting variables generated by a custom action have an associated scope of either AT_BEGIN, NESTED, or AT_END.

The class name (VariableInfo.getClassName) in the returned objects is used to determine the types of the scripting variables. Note that because scripting variables are assigned their values from scoped attributes which cannot be of primitive types, "boxed" types such as java.lang.Integer must be used instead of primitives.

The class name may be a Fully Qualified Class Name, or a short class name.

If a Fully Qualified Class Name is provided, it should refer to a class that should be in the CLASSPATH for the Web Application (see Servlet 2.4 specification - essentially it is WEB-INF/lib and WEB-INF/classes). Failure to be so will lead to a translation-time error.

If a short class name is given in the VariableInfo objects, then the class name must be that of a public class in the context of the import directives of the page where the custom action appears. The class must also be in the CLASSPATH for the Web Application (see Servlet 2.4 specification - essentially it is WEB-INF/lib and WEB-INF/classes). Failure to be so will lead to a translation-time error.

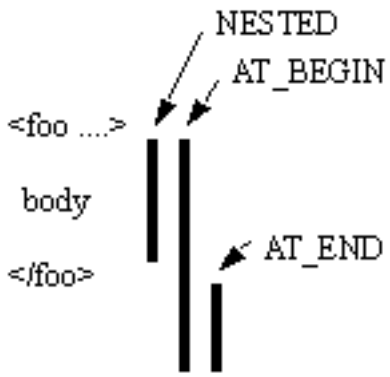
Usage Comments

Frequently a fully qualified class name will refer to a class that is known to the tag library and thus, delivered in the same JAR file as the tag handlers. In most other remaining cases it will refer to a class that is in the platform on which the JSP processor is built (like J2EE). Using fully qualified class names in this manner makes the usage relatively resistant to configuration errors.

A short name is usually generated by the tag library based on some attributes passed through from the custom action user (the author), and it is thus less robust: for instance a missing import directive in the referring JSP page will lead to an invalid short name class and a translation error.

Synchronization Protocol

The result of the invocation on getVariableInfo is an array of VariableInfo objects. Each such object describes a scripting variable by providing its name, its type, whether the variable is new or not, and what its scope is. Scope is best described through a picture:



The JSP 2.0 specification defines the interpretation of 3 values:

- •NESTED, if the scripting variable is available between the start tag and the end tag of the action that defines it.
- •AT_BEGIN, if the scripting variable is available from the start tag of the action that defines it until the end of the scope.
- •AT_END, if the scripting variable is available after the end tag of the action that defines it until the end of the scope.

The scope value for a variable implies what methods may affect its value and thus where synchronization is needed as illustrated by the table below. **Note:** the synchronization of the variable(s) will occur *after* the respective method has been called.

Variable Synchronization Points					
	doStartTag()	**doInitBody()**	**doAfterBody()**	**doEndTag()**	**doTag()**
Tag	AT_BEGIN, NESTED			AT_BEGIN, AT_END	
IterationTag	AT_BEGIN, NESTED		AT_BEGIN, NESTED	AT_BEGIN, AT_END	
BodyTag	AT_BEGIN, NESTED[1]	AT_BEGIN, NESTED[1]	AT_BEGIN, NESTED	AT_BEGIN, AT_END	
SimpleTag					AT_BEGIN, AT_END

[1] Called after doStartTag() if EVAL_BODY_INCLUDE is returned, or after doInitBody() otherwise.

Variable Information in the TLD

Scripting variable information can also be encoded directly for most cases into the Tag Library Descriptor using the <variable> subelement of the <tag> element. See the JSP specification.

JSP.13.9.10.1 Fields

public static final int **AT_BEGIN**

Scope information that scripting variable is visible after start tag.

public static final int **AT_END**

Scope information that scripting variable is visible after end tag.

public static final int **NESTED**

Scope information that scripting variable is visible only within the start/end tags.

JSP.13.9.10.2 Constructors

public **VariableInfo**(java.lang.String varName, java.lang.String className, boolean declare, int scope)

Constructor These objects can be created (at translation time) by the TagExtraInfo instances.

Parameters:
varName - The name of the scripting variable

className - The type of this variable

declare - If true, it is a new variable (in some languages this will require a declaration)

scope - Indication on the lexical scope of the variable

JSP.13.9.10.3 Methods

public java.lang.String **getClassName**()

Returns the type of this variable.

Returns: the type of this variable

public boolean **getDeclare**()

Returns whether this is a new variable. If so, in some languages this will require a declaration.

Returns: whether this is a new variable.

public int **getScope**()

Returns the lexical scope of the variable.

Returns: the lexical scope of the variable, either AT_BEGIN, AT_END, or NESTED.

See Also: public static final int AT_BEGIN, public static final int AT_END, public static final int NESTED

public java.lang.String **getVarName**()

Returns the name of the scripting variable.

Returns: the name of the scripting variable

JSP.13.9.11 TagVariableInfo

Syntax

public class TagVariableInfo

Description

Variable information for a tag in a Tag Library; This class is instantiated from the Tag Library Descriptor file (TLD) and is available only at translation time. This object should be immutable. This information is only available in JSP 1.2 format TLDs or above.

JSP.13.9.11.1 Constructors

public **TagVariableInfo**(java.lang.String nameGiven,
java.lang.String nameFromAttribute, java.lang.String className,
boolean declare, int scope)

Constructor for TagVariableInfo.

Parameters:
nameGiven - value of <name-given>

nameFromAttribute - value of <name-from-attribute>

className - value of <variable-class>

declare - value of <declare>

scope - value of <scope>

JSP.13.9.11.2 Methods

public java.lang.String **getClassName**()

The body of the <variable-class> element.

Returns: The name of the class of the variable or 'java.lang.String' if not defined in the TLD.

public boolean **getDeclare**()

The body of the <declare> element.

Returns: Whether the variable is to be declared or not. If not defined in the TLD, 'true' will be returned.

public java.lang.String **getNameFromAttribute**()

The body of the <name-from-attribute> element. This is the name of an attribute whose (translation-time) value will give the name of the variable. One of <name-given> or <name-from-attribute> is required.

Returns: The attribute whose value defines the variable name

public java.lang.String **getNameGiven**()

The body of the <name-given> element.

Returns: The variable name as a constant

public int **getScope**()

The body of the <scope> element.

Returns: The scope to give the variable. NESTED scope will be returned if not defined in the TLD.

JSP.13.9.12 FunctionInfo

Syntax

public class FunctionInfo

Description

Information for a function in a Tag Library. This class is instantiated from the Tag Library Descriptor file (TLD) and is available only at translation time.

Since: 2.0

JSP.13.9.12.1 Constructors

public **FunctionInfo**(java.lang.String name, java.lang.String klass, java.lang.String signature)

Constructor for FunctionInfo.

Parameters:
name - The name of the function

klass - The class of the function

signature - The signature of the function

JSP.13.9.12.2 Methods

public java.lang.String **getFunctionClass**()

The class of the function.

Returns: The class of the function

public java.lang.String **getFunctionSignature**()

The signature of the function.

Returns: The signature of the function

public java.lang.String **getName**()

The name of the function.

Returns: The name of the function

CHAPTER JSP.14

Expression Language API

This chapter describes the javax.servlet.jsp.el package. The chapter includes content that is generated automatically from javadoc embedded into the actual Java classes and interfaces. This allows the creation of a single, authoritative, specification document.

The javax.servlet.jsp.el package contains a number of classes and interfaces that describe and define programmatic access to the Expression Language evaluator. This API can also be used by an implementation of JSP to evaluate the expressions, but other implementations, like open-coding into Java bytecodes, are allowed. This package is intended to have no dependencies on other portions of the JSP 2.0 specification.

JSP.14.1 Expression Evaluator

Programmatic access to the EL Expression Evaluator is provided through the following types:

- ExpressionEvaluator
- Expression
- FunctionMapper
- VariableResolver

An ExpressionEvaluator object can be obtained from a JspContext object through the getExpressionEvaluator method. An ExpressionEvaluator encapsulates the EL processor. An EL expression provided as a String can then be evaluated directly, or it can be parsed first into an Expression object. The parse step, can be used to factor out the cost of parsing the expression, or even the cost of optimizing the implementation.

The parsing of an expression string is done against a target type, a default prefix (that applies when a function has no prefix), and a FunctionMapper. The

FunctionMapper object maps a prefix and a local name part into a java.lang.reflect.Method object.

The interpretation or evaluation of a parsed expression is done using a VariableResolver object. This object resolves top level object names into Objects. A VariableResolver can be obtained from a JspContext object through the getVariableResolver method.

JSP.14.1.1 ExpressionEvaluator

Syntax

public abstract class ExpressionEvaluator

Description

The abstract base class for an expression-language evaluator. Classes that implement an expression language expose their functionality via this abstract class.

An instance of the ExpressionEvaluator can be obtained via the JspContext / PageContext

The parseExpression() and evaluate() methods must be thread-safe. That is, multiple threads may call these methods on the same ExpressionEvaluator object simultaneously. Implementations should synchronize access if they depend on transient state. Implementations should not, however, assume that only one object of each ExpressionEvaluator type will be instantiated; global caching should therefore be static.

Only a single EL expression, starting with '${' and ending with '}', can be parsed or evaluated at a time. EL expressions cannot be mixed with static text. For example, attempting to parse or evaluate "abc${1+1}def${1+1}ghi" or even "${1+1}${1+1}" will cause an ELException to be thrown.

The following are examples of syntactically legal EL expressions:

- ${person.lastName}
- ${8 * 8}
- ${my:reverse('hello')}

Since: 2.0

JSP.14.1.1.1 Constructors

public **ExpressionEvaluator**()

JSP.14.1.1.2 Methods

public abstract java.lang.Object **evaluate**(java.lang.String expression, java.lang.Class expectedType, VariableResolver vResolver, FunctionMapper fMapper)

Evaluates an expression. This method may perform some syntactic validation and, if so, it should raise an ELParseException error if it encounters syntactic errors. EL evaluation errors should cause an ELException to be raised.

Parameters:
expression - The expression to be evaluated.

expectedType - The expected type of the result of the evaluation

vResolver - A VariableResolver instance that can be used at runtime to resolve the name of implicit objects into Objects.

fMapper - A FunctionMapper to resolve functions found in the expression. It can be null, in which case no functions are supported for this invocation.

Returns: The result of the expression evaluation.

Throws:
ELException - Thrown if the expression evaluation failed.

public abstract Expression **parseExpression**(java.lang.String expression, java.lang.Class expectedType, FunctionMapper fMapper)

Prepare an expression for later evaluation. This method should perform syntactic validation of the expression; if in doing so it detects errors, it should raise an ELParseException.

Parameters:
expression - The expression to be evaluated.

expectedType - The expected type of the result of the evaluation

fMapper - A FunctionMapper to resolve functions found in the expression. It can be null, in which case no functions are supported for this invocation. The ExpressionEvaluator must not hold on to the FunctionMapper reference after returning from parseExpression(). The Expression object returned must invoke the same functions regardless of whether the mappings in the provided FunctionMapper instance change between calling ExpressionEvaluator.parseExpression() and Expression.evaluate().

Returns: The Expression object encapsulating the arguments.

Throws:
ELException - Thrown if parsing errors were found.

JSP.14.1.2 Expression

Syntax

public abstract class Expression

Description

The abstract class for a prepared expression.

An instance of an Expression can be obtained via from an ExpressionEvaluator instance.

An Expression may or not have done a syntactic parse of the expression. A client invoking the evaluate() method should be ready for the case where ELParse-Exception exceptions are raised.

Since: 2.0

JSP.14.1.2.1 Constructors

public **Expression**()

JSP.14.1.2.2 Methods

public abstract java.lang.Object **evaluate**(VariableResolver vResolver)

Evaluates an expression that was previously prepared. In some implementations preparing an expression involves full syntactic validation, but others may not do so. Evaluating the expression may raise an ELParseException as well as other ELExceptions due to run-time evaluation.

Parameters:
vResolver - A VariableResolver instance that can be used at runtime to resolve the name of implicit objects into Objects.

Returns: The result of the expression evaluation.

Throws:
ELException - Thrown if the expression evaluation failed.

JSP.14.1.3 VariableResolver

Syntax

public interface VariableResolver

Description

This class is used to customize the way an ExpressionEvaluator resolves variable references at evaluation time. For example, instances of this class can implement their own variable lookup mechanisms, or introduce the notion of “implicit variables” which override any other variables. An instance of this class should be passed when evaluating an expression.

An instance of this class includes the context against which resolution will happen

Since: 2.0

JSP.14.1.3.1 Methods

public java.lang.Object **resolveVariable**(java.lang.String pName)

Resolves the specified variable. Returns null if the variable is not found.

Parameters:
pName - the name of the variable to resolve

Returns: the result of the variable resolution

Throws:
ELException - if a failure occurred while trying to resolve the given variable

JSP.14.1.4 FunctionMapper

Syntax

public interface FunctionMapper

Description

The interface to a map between EL function names and methods.

Classes implementing this interface may, for instance, consult tag library information to resolve the map.

Since: 2.0

JSP.14.1.4.1 Methods

public java.lang.reflect.Method **resolveFunction**(java.lang.String prefix,
java.lang.String localName)

Resolves the specified local name and prefix into a Java.lang.Method. Returns null if the prefix and local name are not found.

Parameters:
prefix - the prefix of the function, or "" if no prefix.

localName - the short name of the function

Returns: the result of the method mapping. Null means no entry found.

JSP.14.2 Exceptions

The ELException exception is used by the expression language to denote any exception that may arise during the parsing or evaluation of an expression. The ELParseException exception is a subclass of ELException that corresponds to parsing errors

Parsing errors are conveyed as exceptions to simplify the API. It is expected that many JSP containers will use additional mechanisms to parse EL expressions and report their errors - a run-time API cannot provide accurate line-error numbers without additional machinery.

JSP.14.2.1 ELException

Syntax

public class ELException extends java.lang.Exception

Direct Known Subclasses: ELParseException

All Implemented Interfaces: java.io.Serializable

Description

Represents any of the exception conditions that arise during the operation evaluation of the evaluator.

Since: 2.0

JSP.14.2.1.1 Constructors

public **ELException**()

Creates an ELException with no detail message.

public **ELException**(java.lang.String pMessage)

Creates an ELException with the provided detail message.

Parameters:
pMessage - the detail message

public **ELException**(java.lang.String pMessage,
java.lang.Throwable pRootCause)

Creates an ELException with the given detail message and root cause.

Parameters:
pMessage - the detail message

pRootCause - the originating cause of this exception

public **ELException**(java.lang.Throwable pRootCause)

Creates an ELException with the given root cause.

Parameters:
pRootCause - the originating cause of this exception

JSP.14.2.1.2 Methods

public java.lang.Throwable **getRootCause**()

Returns the root cause.

Returns: the root cause of this exception

JSP.14.2.2 ELParseException

Syntax

public class ELParseException extends ELException

All Implemented Interfaces: java.io.Serializable

Description

Represents a parsing error encountered while parsing an EL expression.

Since: 2.0

JSP.14.2.2.1 Constructors

public **ELParseException**()

Creates an ELParseException with no detail message.

public **ELParseException**(java.lang.String pMessage)

Creates an ELParseException with the provided detail message.

Parameters:
pMessage - the detail message

JSP.14.3 Code Fragment

Below is a non-normative code fragment outlining how the APIs can be used.

```
// Get an instance of an ExpressionEvaluator
ExpressionEvaluator ee = myJspContext.getExpressionEvaluator();
VariableResolver vr = myJspContext.getVariableResolver();
FunctionMapper fm; // we don't have a portable implementation yet
// Example of compiling an expression.  See [ISSUE-2]
// Errors detected this way may have higher quality than those
// found with a simple validate() invocation.
ExpressionCompilation ce;
try {
  ce = ee.prepareExpression(expr,
    targetClass,
    fm,
    null // no prefixes
    );
} catch (ELParseException e) {
log (e.getMessage());
}
try {
  ce.evaluate(vr);
} catch (ElException e) {
log (e);
}
```

Part III

The next Appendices provide details.

Appendices B, C and D are normative. Appendices A, E, and F are non-normative.

The Appendices are

- Appendix A - Packaging JSP pages
- Appendix B - Schema for the portion of web.xml owned by the JSP specification
- Appendix C - Schema for the Tag Library Descriptor file.
- Appendix D - Page Character Encoding Detection Algorithm
- Appendix E - Changes
- Appendix F - Glossary of terms

APPENDIX JSP.A

Packaging JSP Pages

This appendix shows two simple examples of packaging a JSP page into a WAR for delivery into a Web container. In the first example, the JSP page is delivered in source form. This is likely to be the most common example. In the second example the JSP page is compiled into a servlet that uses only Servlet 2.4 and JSP 2.0 API calls; the servlet is then packaged into a WAR with a deployment descriptor such that it looks as the original JSP page to any client.

This appendix is non normative. Actually, strictly speaking, the appendix relates more to the Servlet 2.4 capabilities than to the JSP 2.0 capabilities. The appendix is included here as this is a feature that JSP page authors and JSP page authoring tools are interested in.

JSP.A.1A Very Simple JSP Page

We start with a very simple JSP page HelloWorld.jsp.

```
<%@ page info="Example JSP pre-compiled" %>
<p>
Hello World
</p>
```

JSP.A.2The JSP Page Packaged as Source in a WAR File

The JSP page can be packaged into a WAR file by just placing it at location /HelloWorld.jsp the default JSP page extension mapping will pick it up. The web.xml is trivial:

```
<!DOCTYPE webapp
    SYSTEM "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
<webapp>
    <session-config>
        <session-timeout> 1 </session-timeout>
    </session-config>
</webapp>
```

JSP.A.3The Servlet for the Compiled JSP Page

As an alternative, we will show how one can compile the JSP page into a servlet class to run in a JSP container.

The JSP page is compiled into a servlet with some implementation dependent name com.acme._jsp_HelloWorld_XXX_Impl. The servlet code only depends on the JSP 2.0 and Servlet 2.4 APIs, as follows:

```
package com.acme;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

public class _jsp_HelloWorld_XXX_Impl
    extends PlatformDependent_Jsp_Super_Impl
{
    public void _jspInit() {
        // ...
    }

    public void jspDestroy() {
        // ...
    }

    static JspFactory_factory= JspFactory.getDefaultFactory();

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

```
    {
        Object   page= this;
        HttpSessionsession= request.getSession();
        ServletConfigconfig= getServletConfig();
        ServletContextapplication = config.getServletContext();


        PageContextpageContext
            = _factory.getPageContext(this,
                        request,
                        response,
                        (String)NULL,
                        true,
                        JspWriter.DEFAULT_BUFFER,
                        true
                        );

        JspWriterout= pageContext.getOut();
            // page context creates initial JspWriter "out"

        try {
            out.println("<p>");
            out.println("Hello World");
            out.println("</p>");
        } catch (Exception e) {
            pageContext.handlePageException(e);
        } finally {
            _factory.releasePageContext(pageContext);
        }
    }
}
```

JSP.A.4The Web Application Descriptor

The servlet is made to look as a JSP page with the following web.xml:

```
<!DOCTYPE webapp
    SYSTEM "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
<webapp>
    <servlet>
        <servlet-name> HelloWorld </servlet-name>
        <servlet-class>com.acme._jsp_HelloWorld_XXX_Impl</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name> HelloWorld </servlet-name>
        <url-pattern> /HelloWorld.jsp </url-pattern>
    </servlet-mapping>

    <session-config>
        <session-timeout> 1 </session-timeout>
    </session-config>
</webapp>
```

JSP.A.5The WAR for the Compiled JSP Page

Finally everything is packaged together into a WAR:

/WEB-INF/web.xml

/WEB-INF/classes/com/acme/_jsp_HelloWorld_XXX_Impl.class

Note that if the servlet class generated for the JSP page had depended on some support classes, they would have to be included in the WAR.

APPENDIX JSP.B

JSP Elements of web.xml

This appendix describes the JSP elements of the Servlet 2.4 Web Application Deployment Descriptor, which is described using XML Schema. The Servlet 2.4 deployment descriptor schema includes the definitions that appear in this Appendix.

This is the same XML Schema as http://java.sun.com/xml/ns/j2ee/jsp_2_0.xsd, except for some formatting changes to extract comments and make them more readable.

JSP.B.1XML Schema for JSP 2.0 Deployment Descriptor

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://java.sun.com/xml/ns/j2ee"
            xmlns:j2ee="http://java.sun.com/xml/ns/j2ee"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified"
            attributeFormDefault="unqualified"
            version="2.0">

  <xsd:annotation>
    <xsd:documentation>

      This is the XML Schema for the JSP 2.0 deployment descriptor
      types.  The JSP 2.0 schema contains all the special
      structures and datatypes that are necessary to use JSP files
      from a web application.

      The contents of this schema is used by the web-app_2_4.xsd
      file to define JSP specific content.
```

```
      </xsd:documentation>
    </xsd:annotation>

    <xsd:annotation>
      <xsd:documentation>

        The following conventions apply to all J2EE
        deployment descriptor elements unless indicated otherwise.

        - In elements that specify a pathname to a file within the
          same JAR file, relative filenames (i.e., those not
          starting with "/") are considered relative to the root of
          the JAR file's namespace.  Absolute filenames (i.e., those
          starting with "/") also specify names in the root of the
          JAR file's namespace.  In general, relative names are
          preferred.  The exception is .war files where absolute
          names are preferred for consistency with the Servlet API.

      </xsd:documentation>
    </xsd:annotation>

    <xsd:include schemaLocation="j2ee_1_4.xsd"/>

  <!-- ********************************************** -->
    <xsd:complexType name="jsp-configType">

      <xsd:annotation>
        <xsd:documentation>

          The jsp-configType is used to provide global configuration
          information for the JSP files in a web application. It has
          two subelements, taglib and jsp-property-group.

        </xsd:documentation>
      </xsd:annotation>

      <xsd:sequence>
        <xsd:element name="taglib"
                     type="j2ee:taglibType"
                     minOccurs="0"
                     maxOccurs="unbounded"/>
        <xsd:element name="jsp-property-group"
                     type="j2ee:jsp-property-groupType"
                     minOccurs="0"
                     maxOccurs="unbounded"/>
```

```
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

<!-- ************************************************** -->
  <xsd:complexType name="jsp-fileType">

    <xsd:annotation>
      <xsd:documentation>

        The jsp-file element contains the full path to a JSP file
        within the web application beginning with a `/'.

      </xsd:documentation>
    </xsd:annotation>

    <xsd:simpleContent>
      <xsd:restriction base="j2ee:pathType"/>
    </xsd:simpleContent>
  </xsd:complexType>

<!-- ************************************************** -->
  <xsd:complexType name="jsp-property-groupType">

    <xsd:annotation>
      <xsd:documentation>

        The jsp-property-groupType is used to group a number of
        files so they can be given global property information.
        All files so described are deemed to be JSP files.  The
        following additional properties can be described:

            - Control whether EL is ignored
            - Control whether scripting elements are invalid
            - Indicate pageEncoding information.
            - Indicate that a resource is a JSP document (XML)
            - Prelude and Coda automatic includes.

      </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence>
      <xsd:group ref="j2ee:descriptionGroup"/>
      <xsd:element name="url-pattern"
                   type="j2ee:url-patternType"
```

```
              maxOccurs="unbounded"/>
<xsd:element name="el-ignored"
              type="j2ee:true-falseType"
              minOccurs="0">

  <xsd:annotation>
    <xsd:documentation>

      Can be used to easily set the isELIgnored
      property of a group of JSP pages.  By default, the
      EL evaluation is enabled for Web Applications using
      a Servlet 2.4 or greater web.xml, and disabled
      otherwise.

    </xsd:documentation>
  </xsd:annotation>

</xsd:element>
<xsd:element name="page-encoding"
              type="j2ee:string"
              minOccurs="0">

  <xsd:annotation>
    <xsd:documentation>

      The valid values of page-encoding are those of the
      pageEncoding page directive.  It is a
      translation-time error to name different encodings
      in the pageEncoding attribute of the page directive
      of a JSP page and in a JSP configuration element
      matching the page.  It is also a translation-time
      error to name different encodings in the prolog
      or text declaration of a document in XML syntax and
      in a JSP configuration element matching the document.
      It is legal to name the same encoding through
      mulitple mechanisms.

    </xsd:documentation>
  </xsd:annotation>

</xsd:element>
<xsd:element name="scripting-invalid"
              type="j2ee:true-falseType"
              minOccurs="0">
```

```
  <xsd:annotation>
    <xsd:documentation>

      Can be used to easily disable scripting in a
      group of JSP pages.  By default, scripting is
      enabled.

    </xsd:documentation>
  </xsd:annotation>

</xsd:element>
<xsd:element name="is-xml"
             type="j2ee:true-falseType"
             minOccurs="0">

  <xsd:annotation>
    <xsd:documentation>

      If true, denotes that the group of resources
      that match the URL pattern are JSP documents,
      and thus must be interpreted as XML documents.
      If false, the resources are assumed to not
      be JSP documents, unless there is another
      property group that indicates otherwise.

    </xsd:documentation>
  </xsd:annotation>

</xsd:element>
<xsd:element name="include-prelude"
             type="j2ee:pathType"
             minOccurs="0"
             maxOccurs="unbounded">

  <xsd:annotation>
    <xsd:documentation>

      The include-prelude element is a context-relative
      path that must correspond to an element in the
      Web Application.  When the element is present,
      the given path will be automatically included (as
      in an include directive) at the beginning of each
      JSP page in this jsp-property-group.
```

```
            </xsd:documentation>
          </xsd:annotation>

        </xsd:element>
        <xsd:element name="include-coda"
                     type="j2ee:pathType"
                     minOccurs="0"
                     maxOccurs="unbounded">

          <xsd:annotation>
            <xsd:documentation>

              The include-coda element is a context-relative
              path that must correspond to an element in the
              Web Application.  When the element is present,
              the given path will be automatically included (as
              in an include directive) at the end of each
              JSP page in this jsp-property-group.

            </xsd:documentation>
          </xsd:annotation>

        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:complexType>

  <!-- ************************************************** -->
    <xsd:complexType name="taglibType">

      <xsd:annotation>
        <xsd:documentation>

          The taglibType defines the syntax for declaring in
          the deployment descriptor that a tag library is
          available to the application.  This can be done
          to override implicit map entries from TLD files and
          from the container.

        </xsd:documentation>
      </xsd:annotation>

      <xsd:sequence>
        <xsd:element name="taglib-uri"
                     type="j2ee:string">
```

```
        <xsd:annotation>
          <xsd:documentation>

            A taglib-uri element describes a URI identifying a
            tag library used in the web application.  The body
            of the taglib-uri element may be either an
            absolute URI specification, or a relative URI.
            There should be no entries in web.xml with the
            same taglib-uri value.

          </xsd:documentation>
        </xsd:annotation>

      </xsd:element>
      <xsd:element name="taglib-location"
                   type="j2ee:pathType">

        <xsd:annotation>
          <xsd:documentation>

            the taglib-location element contains the location
            (as a resource relative to the root of the web
            application) where to find the Tag Library
            Description file for the tag library.

          </xsd:documentation>
        </xsd:annotation>

      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

</xsd:schema>
```

APPENDIX JSP.C

Tag Library Descriptor Formats

This appendix includes the XML Schema and DTD files for tag library descriptors using each version of the JSP specification (from JSP 1.1 to current). All JSP 2.0 containers are required to be able to parse and accept all TLD formats described in this appendix. The formats are listed in order from most recent to least recent.

JSP.C.1XML Schema for TLD, JSP 2.0

The following is an XML Schema file describing a Tag Library Descriptor in a JSP 2.0 format. This is the same XSD as http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd, except for some formatting changes to extract comments and make them more readable. Some of the types used in this XSD are defined in the J2EE Platform Specification (see Related Documents in the Preface for a link to this specification).

The schema is preceeded by a set of diagrams that graphically illustrate the element structure of the schema. The symbols '+', '*', '|', '(' and ')' have the same meaning as in DTD. In the event of a discrepancy between these diagrams and the schema, the schema prevails.

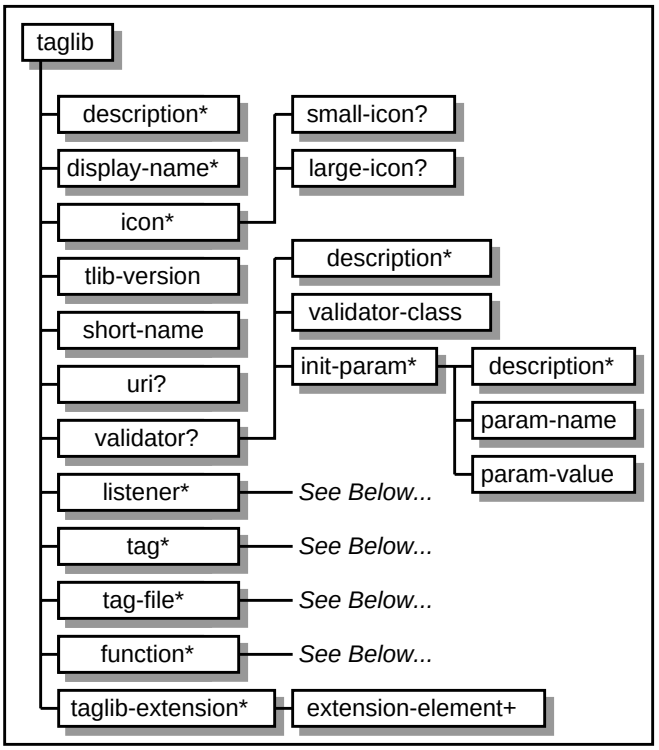


Figure JSP.C-1 TLD Schema Element Structure

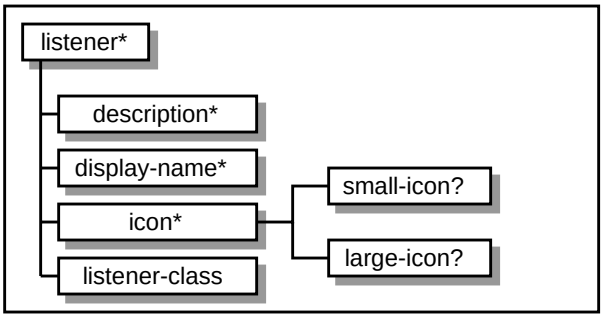


Figure JSP.C-2 TLD Schema Element Structure - listener

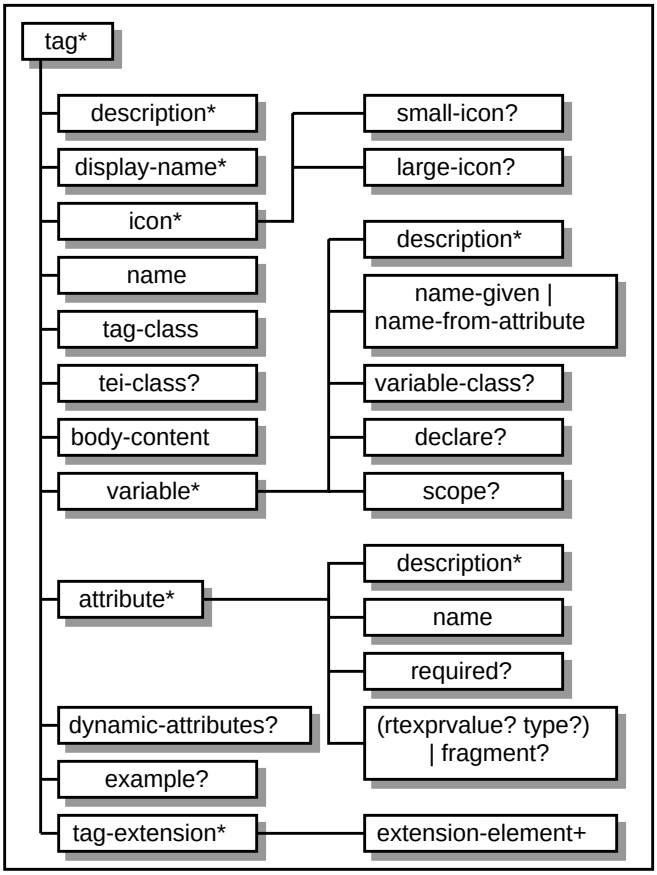


Figure JSP.C-3 TLD Schema Element Structure - tag

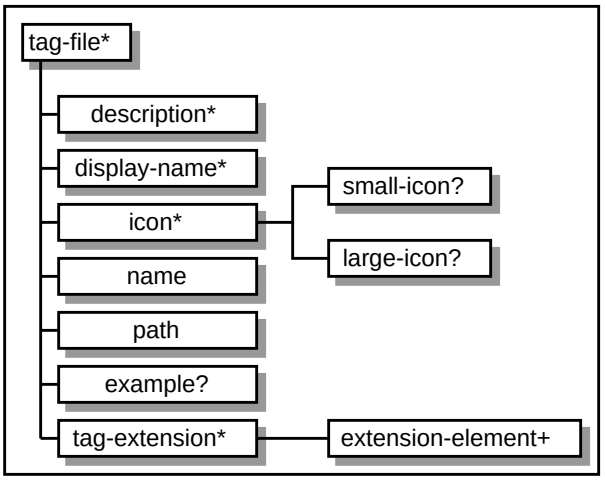


Figure JSP.C-4 TLD Schema Element Structure - tag-file

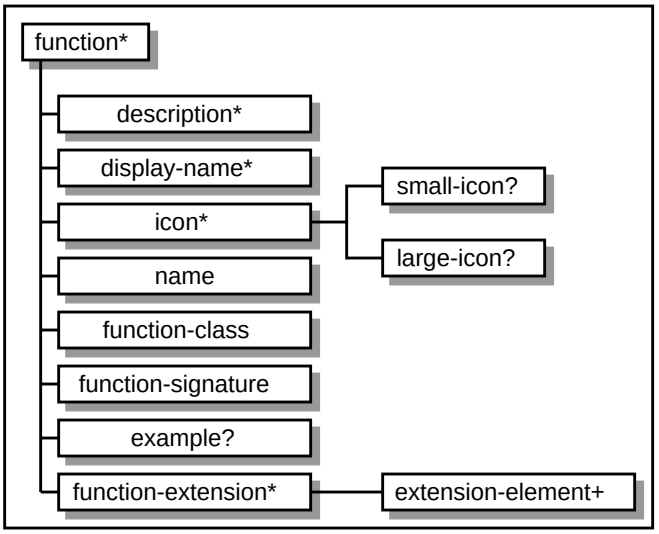


Figure JSP.C-5 TLD Schema Element Structure - function

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
     targetNamespace="http://java.sun.com/xml/ns/j2ee"
     xmlns:j2ee="http://java.sun.com/xml/ns/j2ee"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

```
    elementFormDefault="qualified"
    attributeFormDefault="unqualified"
    version="2.0">

  <xsd:annotation>
    <xsd:documentation>

      This is the XML Schema for the JSP Taglibrary
      descriptor.  All Taglibrary descriptors must
      indicate the tag library schema by using the Taglibrary
      namespace:

      http://java.sun.com/xml/ns/j2ee

      and by indicating the version of the schema by
      using the version element as shown below:

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

      The instance documents may indicate the published
      version of the schema using xsi:schemaLocation attribute
      for J2EE namespace with the following location:

      http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd

    </xsd:documentation>
  </xsd:annotation>

  <xsd:include schemaLocation="j2ee_1_4.xsd"/>

<!-- **************************************************** -->
  <xsd:element name="taglib" type="j2ee:tldTaglibType">

    <xsd:annotation>
      <xsd:documentation>

        The taglib tag is the document root.
        The definition of taglib is provided
        by the tldTaglibType.
```

```
      </xsd:documentation>
    </xsd:annotation>

    <xsd:unique name="tag-name-uniqueness">

      <xsd:annotation>
        <xsd:documentation>

          The taglib element contains, among other things, tag and
          tag-file elements.
          The name subelements of these elements must each be unique.

        </xsd:documentation>
      </xsd:annotation>

      <xsd:selector xpath="j2ee:tag|j2ee:tag-file"/>
      <xsd:field    xpath="j2ee:name"/>
    </xsd:unique>
    <xsd:unique name="function-name-uniqueness">

      <xsd:annotation>
        <xsd:documentation>

          The taglib element contains function elements.
          The name subelements of these elements must each be unique.

        </xsd:documentation>
      </xsd:annotation>

      <xsd:selector xpath="j2ee:function"/>
      <xsd:field    xpath="j2ee:name"/>
    </xsd:unique>
  </xsd:element>

<!-- ************************************************** -->
  <xsd:complexType name="body-contentType">

    <xsd:annotation>
      <xsd:documentation>

        Specifies the type of body that is valid for a tag.
        This value is used by the JSP container to validate
        that a tag invocation has the correct body syntax and
        by page composition tools to assist the page author
        in providing a valid tag body.
```

```
        There are currently four values specified:

        tagdependent    The body of the tag is interpreted by the tag
                        implementation itself, and is most likely
                        in a different "language", e.g embedded SQL
                        statements.

        JSP             The body of the tag contains nested JSP
                        syntax.

        empty           The body must be empty

        scriptless      The body accepts only template text, EL
                        Expressions, and JSP action elements.  No
                        scripting elements are allowed.

      </xsd:documentation>
    </xsd:annotation>

    <xsd:simpleContent>
      <xsd:restriction base="j2ee:string">
        <xsd:enumeration value="tagdependent"/>
        <xsd:enumeration value="JSP"/>
        <xsd:enumeration value="empty"/>
        <xsd:enumeration value="scriptless"/>
      </xsd:restriction>
    </xsd:simpleContent>
  </xsd:complexType>

<!-- ************************************************* -->
  <xsd:complexType name="extensibleType" abstract="true">

    <xsd:annotation>
      <xsd:documentation>

        The extensibleType is an abstract base type that is used to
        define the type of extension-elements. Instance documents
        must substitute a known type to define the extension by
        using xsi:type attribute to define the actual type of
        extension-elements.

      </xsd:documentation>
    </xsd:annotation>
```

```
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

<!-- ************************************************* -->
  <xsd:complexType name="functionType">

    <xsd:annotation>
      <xsd:documentation>

        The function element is used to provide information on each
        function in the tag library that is to be exposed to the EL.

        The function element may have several subelements defining:

        description         Optional tag-specific information

        display-name        A short name that is intended to be
                            displayed by tools

        icon                Optional icon element that can be used
                            by tools

        name                A unique name for this function

        function-class      Provides the name of the Java class that
                            implements the function

        function-signature  Provides the signature, as in the Java
                            Language Specification, of the Java
                            method that is to be used to implement
                            the function.

        example             Optional informal description of an
                            example of a use of this function

        function-extension  Zero or more extensions that provide extra
                            information about this function, for tool
                            consumption

      </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence>
      <xsd:group ref="j2ee:descriptionGroup"/>
      <xsd:element name="name"
```

```
                type="j2ee:tld-canonical-nameType">

    <xsd:annotation>
      <xsd:documentation>

        A unique name for this function.

      </xsd:documentation>
    </xsd:annotation>

  </xsd:element>
  <xsd:element name="function-class"
                type="j2ee:fully-qualified-classType">

    <xsd:annotation>
      <xsd:documentation>

        Provides the fully-qualified class name of the Java
        class containing the static method that implements
        the function.

      </xsd:documentation>
    </xsd:annotation>

  </xsd:element>
  <xsd:element name="function-signature"
                type="j2ee:string">

    <xsd:annotation>
      <xsd:documentation>

        Provides the signature, of the static Java method that is
        to be used to implement the function.  The syntax of the
        function-signature element is as follows:

            FunctionSignature ::= ReturnType S MethodName S?
                                  '(' S? Parameters? S? ')'

            ReturnType        ::= Type

            MethodName        ::= Identifier

            Parameters        ::=   Parameter
                                | ( Parameter S? ',' S? Parameters )
```

```
            Parameter         ::= Type

            Where:

                * Type is a basic type or a fully qualified
                  Java class name (including package name),
                  as per the 'Type' production in the Java
                  Language Specification, Second Edition,
                  Chapter 18.

                * Identifier is a Java identifier, as per
                  the 'Identifier' production in the Java
                  Language Specification, Second
                  Edition, Chapter 18.

          Example:

          java.lang.String nickName( java.lang.String, int )

        </xsd:documentation>
      </xsd:annotation>

    </xsd:element>
    <xsd:element name="example"
                 type="j2ee:xsdStringType"
                 minOccurs="0">

      <xsd:annotation>
        <xsd:documentation>

          The example element contains an informal description
          of an example of the use of this function.

        </xsd:documentation>
      </xsd:annotation>

    </xsd:element>
    <xsd:element name="function-extension"
                 type="j2ee:tld-extensionType"
                 minOccurs="0"
                 maxOccurs="unbounded">

      <xsd:annotation>
        <xsd:documentation>
```

```
          Function extensions are for tool use only and must not affect
            the behavior of a container.

          </xsd:documentation>
        </xsd:annotation>

      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

<!-- ************************************************* -->
  <xsd:complexType name="tagFileType">

    <xsd:annotation>
      <xsd:documentation>

        Defines an action in this tag library that is implemented
        as a .tag file.

        The tag-file element has two required subelements:

        description       Optional tag-specific information

        display-name      A short name that is intended to be
                          displayed by tools

        icon              Optional icon element that can be used
                          by tools

        name              The unique action name

        path              Where to find the .tag file implementing this
                           action, relative to the root of the web
                          application or the root of the JAR file for a
                           tag library packaged in a JAR.  This must
                           begin with /WEB-INF/tags if the .tag file
                          resides in the WAR, or /META-INF/tags if the
                           .tag file resides in a JAR.

        example           Optional informal description of an
                          example of a use of this tag

        tag-extension     Zero or more extensions that provide extra
                          information about this tag, for tool
```

```
                        consumption

      </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence>
      <xsd:group ref="j2ee:descriptionGroup"/>
      <xsd:element name="name"
                   type="j2ee:tld-canonical-nameType"/>
      <xsd:element name="path"
                   type="j2ee:pathType"/>
      <xsd:element name="example"
                   type="j2ee:xsdStringType"
                   minOccurs="0">

        <xsd:annotation>
          <xsd:documentation>

            The example element contains an informal description
            of an example of the use of a tag.

          </xsd:documentation>
        </xsd:annotation>

      </xsd:element>
      <xsd:element name="tag-extension"
                   type="j2ee:tld-extensionType"
                   minOccurs="0"
                   maxOccurs="unbounded">

        <xsd:annotation>
          <xsd:documentation>

            Tag extensions are for tool use only and must not affect
            the behavior of a container.

          </xsd:documentation>
        </xsd:annotation>

      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>
```

```
<!-- ************************************************** -->
  <xsd:complexType name="tagType">

    <xsd:annotation>
      <xsd:documentation>

        The tag defines a unique tag in this tag library.  It has one
        attribute, id.

        The tag element may have several subelements defining:

        description       Optional tag-specific information

        display-name      A short name that is intended to be
                          displayed by tools

        icon              Optional icon element that can be used
                          by tools

        name              The unique action name

        tag-class         The tag handler class implementing
                          javax.servlet.jsp.tagext.JspTag

        tei-class         An optional subclass of
                          javax.servlet.jsp.tagext.TagExtraInfo

        body-content      The body content type

        variable          Optional scripting variable information

        attribute         All attributes of this action that are
                          evaluated prior to invocation.

        dynamic-attributes Whether this tag supports additional
                           attributes with dynamic names.  If
                           true, the tag-class must implement the
                           javax.servlet.jsp.tagext.DynamicAttributes
                           interface.  Defaults to false.

        example           Optional informal description of an
                          example of a use of this tag

        tag-extension     Zero or more extensions that provide extra
```

```
                        information about this tag, for tool
                        consumption

    </xsd:documentation>
  </xsd:annotation>

  <xsd:sequence>
    <xsd:group ref="j2ee:descriptionGroup"/>
    <xsd:element name="name"
                 type="j2ee:tld-canonical-nameType"/>
    <xsd:element name="tag-class"
                 type="j2ee:fully-qualified-classType">

      <xsd:annotation>
        <xsd:documentation>

          Defines the subclass of javax.serlvet.jsp.tagext.JspTag
          that implements the request time semantics for
          this tag. (required)

        </xsd:documentation>
      </xsd:annotation>

    </xsd:element>
    <xsd:element name="tei-class"
                 type="j2ee:fully-qualified-classType"
                 minOccurs="0">

      <xsd:annotation>
        <xsd:documentation>

        Defines the subclass of javax.servlet.jsp.tagext.TagExtraInfo
          for this tag. (optional)

          If this is not given, the class is not consulted at
          translation time.

        </xsd:documentation>
      </xsd:annotation>

    </xsd:element>
    <xsd:element name="body-content"
                 type="j2ee:body-contentType">
```

```
      <xsd:annotation>
        <xsd:documentation>

          Specifies the format for the body of this tag.
          The default in JSP 1.2 was "JSP" but because this
          is an invalid setting for simple tag handlers, there
          is no longer a default in JSP 2.0.  A reasonable
          default for simple tag handlers is "scriptless" if
          the tag can have a body.

        </xsd:documentation>
      </xsd:annotation>

    </xsd:element>
    <xsd:element name="variable"
                 type="j2ee:variableType"
                 minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="attribute"
                 type="j2ee:tld-attributeType"
                 minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="dynamic-attributes"
                 type="j2ee:generic-booleanType"
                 minOccurs="0"/>
    <xsd:element name="example"
                 type="j2ee:xsdStringType"
                 minOccurs="0">

      <xsd:annotation>
        <xsd:documentation>

          The example element contains an informal description
          of an example of the use of a tag.

        </xsd:documentation>
      </xsd:annotation>

    </xsd:element>
    <xsd:element name="tag-extension"
                 type="j2ee:tld-extensionType"
                 minOccurs="0"
                 maxOccurs="unbounded">

      <xsd:annotation>
        <xsd:documentation>
```

```
            Tag extensions are for tool use only and must not affect
            the behavior of a container.

          </xsd:documentation>
        </xsd:annotation>

      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

<!-- ************************************************* -->
  <xsd:complexType name="tld-attributeType">

    <xsd:annotation>
      <xsd:documentation>

        The attribute element defines an attribute for the nesting
        tag.  The attributre element may have several subelements
        defining:

        description     a description of the attribute

        name            the name of the attribute

        required        whether the attribute is required or
                        optional

        rtexprvalue     whether the attribute is a runtime attribute

        type            the type of the attributes

        fragment        whether this attribute is a fragment

      </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence>
      <xsd:element name="description"
                   type="j2ee:descriptionType"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="name"
                   type="j2ee:java-identifierType"/>
      <xsd:element name="required"
                   type="j2ee:generic-booleanType"
```

```
                minOccurs="0">

      <xsd:annotation>
        <xsd:documentation>

          Defines if the nesting attribute is required or
          optional.

          If not present then the default is "false", i.e
          the attribute is optional.

        </xsd:documentation>
      </xsd:annotation>

    </xsd:element>
    <xsd:choice>
      <xsd:sequence>
        <xsd:element name="rtexprvalue"
                     type="j2ee:generic-booleanType"
                     minOccurs="0">

          <xsd:annotation>
            <xsd:documentation>

              Defines if the nesting attribute can have scriptlet
              expressions as a value, i.e the value of the
              attribute may be dynamically calculated at request
              time, as opposed to a static value determined at
              translation time.

              If not present then the default is "false", i.e the
              attribute has a static value

            </xsd:documentation>
          </xsd:annotation>

        </xsd:element>
        <xsd:element name="type"
                     type="j2ee:fully-qualified-classType"
                     minOccurs="0">

          <xsd:annotation>
            <xsd:documentation>

              Defines the Java type of the attributes value.  For
```

```
                static values (those determined at translation time)
                the type is always java.lang.String.

              </xsd:documentation>
            </xsd:annotation>

          </xsd:element>
        </xsd:sequence>
        <xsd:element name="fragment"
                     type="j2ee:generic-booleanType"
                     minOccurs="0">

          <xsd:annotation>
            <xsd:documentation>

              "true" if this attribute is of type
              javax.jsp.tagext.JspFragment, representing dynamic
              content that can be re-evaluated as many times
              as needed by the tag handler.  If omitted or "false",
              the default is still type="java.lang.String"

            </xsd:documentation>
          </xsd:annotation>

        </xsd:element>
      </xsd:choice>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

<!-- ************************************************** -->
  <xsd:complexType name="tld-canonical-nameType">

    <xsd:annotation>
      <xsd:documentation>

        Defines the canonical name of a tag or attribute being
        defined.

        The name must conform to the lexical rules for an NMTOKEN.

      </xsd:documentation>
    </xsd:annotation>

    <xsd:simpleContent>
```

```
      <xsd:restriction base="j2ee:xsdNMTOKENType"/>
    </xsd:simpleContent>
  </xsd:complexType>

<!-- ************************************************** -->
  <xsd:complexType name="tld-extensionType">

    <xsd:annotation>
      <xsd:documentation>

        The tld-extensionType is used to indicate
        extensions to a specific TLD element.

        It is used by elements to designate an extension block
        that is targeted to a specific extension designated by
        a set of extension elements that are declared by a
        namespace. The namespace identifies the extension to
        the tool that processes the extension.

        The type of the extension-element is abstract. Therefore,
        a concrete type must be specified by the TLD using
        xsi:type attribute for each extension-element.

      </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence>
      <xsd:element name="extension-element"
                   type="j2ee:extensibleType"
                   maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="namespace"
                   use="required"
                   type="xsd:anyURI"/>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

<!-- ************************************************** -->
  <xsd:complexType name="tldTaglibType">

    <xsd:annotation>
      <xsd:documentation>

        The taglib tag is the document root, it defines:
```

```
    description     a simple string describing the "use" of this
                    taglib, should be user discernable

    display-name    the display-name element contains a
                    short name that is intended to be displayed
                    by tools

    icon            optional icon that can be used by tools

    tlib-version    the version of the tag library implementation

    short-name      a simple default short name that could be
                    used by a JSP authoring tool to create
                    names with a mnemonic value; for example,
                    the it may be used as the prefered prefix
                    value in taglib directives

    uri             a uri uniquely identifying this taglib

    validator       optional TagLibraryValidator information

    listener        optional event listener specification

    tag             tags in this tag library

    tag-file        tag files in this tag library

    function        zero or more EL functions defined in this
                    tag library

    taglib-extension zero or more extensions that provide extra
                    information about this taglib, for tool
                    consumption

  </xsd:documentation>
</xsd:annotation>

<xsd:sequence>
  <xsd:group ref="j2ee:descriptionGroup"/>
  <xsd:element name="tlib-version"
               type="j2ee:dewey-versionType">

    <xsd:annotation>
      <xsd:documentation>
```

```
        Describes this version (number) of the taglibrary.
        It is described as a dewey decimal.

      </xsd:documentation>
    </xsd:annotation>

  </xsd:element>
  <xsd:element name="short-name"
               type="j2ee:tld-canonical-nameType">

    <xsd:annotation>
      <xsd:documentation>

        Defines a simple default name that could be used by
        a JSP authoring tool to create names with a
        mnemonicvalue; for example, it may be used as the
        preferred prefix value in taglib directives.  Do
        not use white space, and do not start with digits
        or underscore.

      </xsd:documentation>
    </xsd:annotation>

  </xsd:element>
  <xsd:element name="uri"
               type="j2ee:xsdAnyURIType"
               minOccurs="0">

    <xsd:annotation>
      <xsd:documentation>

        Defines a public URI that uniquely identifies this
        version of the taglibrary.  Leave it empty if it
        does not apply.

      </xsd:documentation>
    </xsd:annotation>

  </xsd:element>
  <xsd:element name="validator"
               type="j2ee:validatorType"
               minOccurs="0">
  </xsd:element>
  <xsd:element name="listener"
               type="j2ee:listenerType"
```

```
                minOccurs="0" maxOccurs="unbounded">
      </xsd:element>
      <xsd:element name="tag"
                type="j2ee:tagType"
                minOccurs="0"
                maxOccurs="unbounded"/>
      <xsd:element name="tag-file"
                type="j2ee:tagFileType"
                minOccurs="0"
                maxOccurs="unbounded"/>
      <xsd:element name="function"
                type="j2ee:functionType"
                minOccurs="0"
                maxOccurs="unbounded"/>
      <xsd:element name="taglib-extension"
                type="j2ee:tld-extensionType"
                minOccurs="0"
                maxOccurs="unbounded">

        <xsd:annotation>
          <xsd:documentation>

            Taglib extensions are for tool use only and must not affect
             the behavior of a container.

          </xsd:documentation>
        </xsd:annotation>

      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="version"
                type="j2ee:dewey-versionType"
                fixed="2.0"
                use="required">

      <xsd:annotation>
        <xsd:documentation>

          Describes the JSP version (number) this taglibrary
          requires in order to function (dewey decimal)

        </xsd:documentation>
      </xsd:annotation>

    </xsd:attribute>
```

```
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

<!-- **************************************************** -->
  <xsd:complexType name="validatorType">

    <xsd:annotation>
      <xsd:documentation>

        A validator that can be used to validate
        the conformance of a JSP page to using this tag library is
        defined by a validatorType.

      </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence>
      <xsd:element name="description"
                   type="j2ee:descriptionType"
                   minOccurs="0"
                   maxOccurs="unbounded"/>
      <xsd:element name="validator-class"
                   type="j2ee:fully-qualified-classType">

        <xsd:annotation>
          <xsd:documentation>

            Defines the TagLibraryValidator class that can be used
            to validate the conformance of a JSP page to using this
            tag library.

          </xsd:documentation>
        </xsd:annotation>

      </xsd:element>
      <xsd:element name="init-param"
                   type="j2ee:param-valueType"
                   minOccurs="0" maxOccurs="unbounded">

        <xsd:annotation>
          <xsd:documentation>

            The init-param element contains a name/value pair as an
            initialization param.
```

```
          </xsd:documentation>
        </xsd:annotation>

      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

<!-- ************************************************** -->
  <xsd:complexType name="variable-scopeType">

    <xsd:annotation>
      <xsd:documentation>

        This type defines scope of the scripting variable.  See
        TagExtraInfo for details.  The allowed values are,
        "NESTED", "AT_BEGIN" and "AT_END".

      </xsd:documentation>
    </xsd:annotation>

    <xsd:simpleContent>
      <xsd:restriction base="j2ee:string">
        <xsd:enumeration value="NESTED"/>
        <xsd:enumeration value="AT_BEGIN"/>
        <xsd:enumeration value="AT_END"/>
      </xsd:restriction>
    </xsd:simpleContent>
  </xsd:complexType>

<!-- ************************************************** -->
  <xsd:complexType name="variableType">

    <xsd:annotation>
      <xsd:documentation>

        The variableType provides information on the scripting
        variables defined by using this tag.  It is a (translation
        time) error for a tag that has one or more variable
        subelements to have a TagExtraInfo class that returns a
        non-null value from a call to getVariableInfo().

        The subelements of variableType are of the form:

        description              Optional description of this
```

```
                              variable

    name-given                The variable name as a constant

    name-from-attribute       The name of an attribute whose
                              (translation time) value will
                              give the name of the
                              variable.  One of name-given or
                              name-from-attribute is required.

    variable-class            Name of the class of the variable.
                              java.lang.String is default.

    declare                   Whether the variable is declared
                              or not.  True is the default.

    scope                     The scope of the scripting varaible
                              defined.  NESTED is default.

  </xsd:documentation>
</xsd:annotation>

<xsd:sequence>
  <xsd:element name="description"
               type="j2ee:descriptionType"
               minOccurs="0" maxOccurs="unbounded"/>
  <xsd:choice>
    <xsd:element name="name-given"
                 type="j2ee:java-identifierType">

      <xsd:annotation>
        <xsd:documentation>

          The name for the scripting variable.

        </xsd:documentation>
      </xsd:annotation>

    </xsd:element>
    <xsd:element name="name-from-attribute"
                 type="j2ee:java-identifierType">

      <xsd:annotation>
        <xsd:documentation>
```

```
        The name of an attribute whose
        (translation-time) value will give the name of
        the variable.

      </xsd:documentation>
    </xsd:annotation>

  </xsd:element>
</xsd:choice>
<xsd:element name="variable-class"
             type="j2ee:fully-qualified-classType"
             minOccurs="0">

  <xsd:annotation>
    <xsd:documentation>

      The optional name of the class for the scripting
      variable.  The default is java.lang.String.

    </xsd:documentation>
  </xsd:annotation>

</xsd:element>
<xsd:element name="declare"
             type="j2ee:generic-booleanType"
             minOccurs="0">

  <xsd:annotation>
    <xsd:documentation>

      Whether the scripting variable is to be defined
      or not.  See TagExtraInfo for details.  This
      element is optional and "true" is the default.

    </xsd:documentation>
  </xsd:annotation>

</xsd:element>
<xsd:element name="scope"
             type="j2ee:variable-scopeType"
             minOccurs="0">

  <xsd:annotation>
    <xsd:documentation>
```

```
            The element is optional and "NESTED" is the default.

          </xsd:documentation>
        </xsd:annotation>

      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

</xsd:schema>
```

JSP.C.2DTD for TLD, JSP 1.2

The following is a DTD describing a Tag Library Descriptor file in JSP 1.2 format. This is the same DTD as "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd", except for some formatting changes to extract comments and make them more readable:

```
<!--
This is the DTD defining the JavaServer Pages 1.2 Tag Library descriptor (.tld)
(XML) file format/syntax.
A Tag Library is a JAR file containing a valid instance of a Tag Library Descriptor
file, along with the appropriate implementation classes and other resources re-
quired to implement the actions defined therein. When deployed inside a JAR file,
the tag library descriptor files must be in the META-INF directory, or a subdirec-
tory of it. When deployed directly into a web application, the tag library descriptor
files must always be in the WEB-INF directory, or some subdirectory of it.
Packaged tag libraries must have at least one tag library descriptor file. The JSP
1.1 specification allowed for only a single TLD, in META-INF/taglib.tld, but in JSP
1.2 multiple tag libraries are allowed.
Use is subject to license terms.
-->

<!NOTATION WEB-JSPTAGLIB.1_2 PUBLIC “-//Sun Microsystems, Inc.//DTD
JSP Tag Library 1.2//EN”>

<!--
All JSP 1.2 tag library descriptors must include a DOCTYPE of the following form:
```

```
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library
1.2//EN" "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
-->

<!--
The taglib element is the document root, it defines:

tlib-version      the version of the tag library implementation
jsp-version       the version of JSP the tag library depends upon
short-name        a simple default name that could be used by a JSP authoring
tool to create names with a mnemonic value; for example, the it may be used as
the prefered prefix value in taglib directives
uri               a uri uniquely identifying this taglib
display-name      the display-name element contains a short name that is intend-
ed to be displayed by tools
small-icon        optional small-icon that can be used by tools
large-icon        optional large-icon that can be used by tools
description       a simple string describing the “use” of this taglib, should be user
discernable
validator         optional TagLibraryValidator information
listener          optional event listener specification
-->

<!ELEMENT taglib (tlib-version, jsp-version, short-name, uri?, display-name?,
small-icon?, large-icon?, description?, validator?, listener*, tag+) >

<!ATTLIST taglib
    id ID #IMPLIED
    xmlns CDATA #FIXED “http://java.sun.com/JSP/TagLibraryDescriptor”>
<!--
The value of the tlib-version element describes this version (number) of the tagl-
ibrary. This element is mandatory.

#PCDATA ::= [0-9]*{ “.”[0-9] }0..3
-->

<!ELEMENT tlib-version (#PCDATA)
```

```
<!--
The value of the jsp-version element describes the JSP version (number) this
taglibrary requires in order to function. This element is mandatory. The value that
should be used for JSP 1.2 is "1.2" (no quotes).

#PCDATA ::= [0-9]*{ “.”[0-9] }0..3
-->

<!ELEMENT jsp-version (#PCDATA) >

<!--
The value of the short-name element is a name that could be used by a JSP au-
thoring tool to create names with a mnemonic value; for example, it may be used
as the prefered prefix value in taglib directives.

Do not use white space, and do not start with digits or underscore.

#PCDATA ::= NMTOKEN
-->

<!ELEMENT short-name (#PCDATA) >

<!--
The value of the uri element is a public URI that uniquely identifies the exact se-
mantics of this taglibrary.
-->

<!ELEMENT uri (#PCDATA) >

<!--
The value of the description element is an arbitrary text string describing the tag
library.
-->

<!ELEMENT description(#PCDATA) >

<!--
The validator element provides information on an optional validator that can be
used to validate the conformance of a JSP page to using this tag library.
-->

<!ELEMENT validator (validator-class, init-param*, description?) >
```

```
<!--
The validator-class element defines the TagLibraryValidator class that can be
used to validate the conformance of a JSP page to using this tag library.
-->

<!ELEMENT validator-class (#PCDATA) >

<!--
The init-param element contains a name/value pair as an initialization param.
-->

<!ELEMENT init-param (param-name, param-value, description?)>

<!--
The param-name element contains the name of a parameter.
-->

<!ELEMENT param-name (#PCDATA)>

<!--
The param-value element contains the value of a parameter.
-->

<!ELEMENT param-value (#PCDATA)>

<!--
The listener element defines an optional event listener object to be instantiated
and registered automatically.
-->

<!ELEMENT listener (listener-class) >

<!--
The listener-class element declares a class in the application that must be regis-
tered as a web application listener bean.

See the Servlet 2.3 specification for details.
-->

<!ELEMENT listener-class (#PCDATA) >
```

```
<!--
The tag element defines an action in this tag library. The tag element has one at-
tribute, id.
The tag element may have several subelements defining:
name            The unique action name
tag-class       The tag handler class implementing javax.servlet.jsp.tagext.Tag
tei-class       An optional subclass of javax.servlet.jsp.tagext.TagExtraInfo
body-content    The body content type
display-name    A short name that is intended to be displayed by tools
small-icon      Optional small-icon that can be used by tools
large-icon      Optional large-icon that can be used by tools
description     Optional tag-specific information
variable        Optional scripting variable information
attribute       All attributes of this action
example         Optional informal description of an example of a use of this ac-
tion.
-->

<!ELEMENT tag (name, tag-class, tei-class?, body-content?, display-name?,
small-icon?, large-icon?, description?, variable*, attribute*, example?) >

<!--
The tag-class element indicates the subclass of javax.serlvet.jsp.tagext.Tag that
implements the request time semantics for this tag. This element is required.

#PCDATA ::= fully qualified Java class name
-->

<!ELEMENT tag-class (#PCDATA) >

<!--
The tei-class element indicates the subclass of javax.servlet.jsp.tagext.TagEx-
traInfo for this tag. The class is instantiated at translation time. This element is
optional.

#PCDATA ::= fully qualified Java class name
-->

<!ELEMENT tei-class (#PCDATA) >
```

```
<!--
The body-content element provides provides information on the content of the
body of this tag. This element is primarily intended for use by page composition
tools.
There are currently three values specified:

tagdependent    The body of the tag is interpreted by the tag implementation it-
self, and is most likely in a different “langage”, e.g embedded SQL statements.
JSP             The body of the tag contains nested JSP syntax
empty           The body must be empty
This element is optional; the default value is JSP

#PCDATA ::=  tagdependent | JSP | empty
-->

<!ELEMENT body-content (#PCDATA) >

<!--
The display-name element contains a short name that is intended to be displayed
by tools.
-->

<!ELEMENT display-name (#PCDATA) >

<!--
The large-icon element contains the name of a file containing a large (32 x 32)
icon image. The icon can be used by tools. The file name is a relative path within
the tag library.
The image must be either in the JPEG or GIF format, and the file name must end
with the suffix “.jpg” or “.gif” respectively.
-->

<!ELEMENT large-icon (#PCDATA) >

<!--
The small-icon element contains the name of a file containing a small (16 x 16)
icon image. The icon can be used by tools. The file name is a relative path within
the tag library.
The image must be either in the JPEG or GIF format, and the file name must end
with the suffix “.jpg” or “.gif” respectively.
-->

<!ELEMENT small-icon (#PCDATA) >
```

```
<!--
The example element provides an informal description of an example of the use
of a tag.
-->

<!ELEMENT example (#PCDATA) >

<!--
The variable element provides information on the scripting variables defined by
this tag.

It is a (translation time) error for an action that has one or more variable subele-
ments to have a TagExtraInfo class that returns a non-null object.

The subelements of variable are of the form:

name-given            The variable name as a constant
name-from-attribute   The name of an attribute whose (translation time) value will
give the name of the variable.  One of name-given or name-from-attribute is re-
quired.
variable-class        Name of the class of the variable. java.lang.String is default.
declare                Whether the variable is declared or not. True is the default.
scope                 The scope of the scripting variable defined.  NESTED is de-
fault.
-->

<!ELEMENT variable ( (name-given | name-from-attribute), variable-class?, de-
clare?, scope?, description?) >

<!--
The name-given element provides the name for the scripting variable.

One of name-given or name-from-attribute is required.
-->

<!ELEMENT name-given (#PCDATA) >

<!--
The value of the name-from-attribute element is the name of an attribute whose
(translation-time) value will give the name of the variable.

One of name-given or name-from-attribute is required.
-->

<!ELEMENT name-from-attribute (#PCDATA) >
```

```
<!--
The variable-class element is the name of the class for the scripting variable.

This element is optional; the default is java.lang.String.
-->

<!ELEMENT variable-class (#PCDATA) >

<!--
The value of the declare element indicates whether the scripting variable is to be
defined or not. See TagExtraInfo for details.

This element is optional and is the default is true.
-->

<!ELEMENT declare (#PCDATA) >

<!--
The value of the scope element describes the scope of the scripting variable.

See TagExtraInfo for details.

This element is optional and the default value is the string “NESTED”. The other
legal values are “AT_BEGIN” and “AT_END”.
-->

<!ELEMENT scope (#PCDATA) >

<!--
The attribute element defines an attribute for the nesting tag.

The attributre element may have several subelements defining:
name          the name of the attribute
attribute     whether the attribute is required or optional
rtexpravalue  whether the attribute is a runtime attribute
type          the type of the attributes
description   a description of the attribute
-->

<!ELEMENT attribute (name, required? , rtexprvalue?, type?, description?) >
```

```
<!--
The name element defines the canonical name of a tag or attribute being defined

#PCDATA ::= NMTOKEN
-->

<!ELEMENT name(#PCDATA) >

<!--
The value of the required element indicates if the nesting attribute is required or
optional. This attribute is optional and its default value is false.

#PCDATA ::= true | false | yes | no
-->

<!ELEMENT required    (#PCDATA) >

<!--
The value of the rtexpvalue element indicates if the value of the attribute may be
dynamically calculated at request time, as opposed to a static value determined
at translation time. This attribute is optional and its default value is false

#PCDATA ::= true | false | yes | no
-->

<!ELEMENT rtexprvalue (#PCDATA) >

<!--
The value of the type element describes the Java type of the attributes value.

For static values (those determined at translation time) the type is always ja-
va.lang.String.
-->

<!ELEMENT type (#PCDATA) >

<!-- ID attributes -->

<!ATTLIST tlib-version id ID #IMPLIED>

<!ATTLIST jsp-version id ID #IMPLIED>

<!ATTLIST short-name id ID #IMPLIED>

<!ATTLIST uri id ID #IMPLIED>
```

```
<!ATTLIST description id ID #IMPLIED>

<!ATTLIST example id ID #IMPLIED>

<!ATTLIST tag id ID #IMPLIED>

<!ATTLIST tag-class id ID #IMPLIED>

<!ATTLIST tei-class id ID #IMPLIED>

<!ATTLIST body-content id ID #IMPLIED>

<!ATTLIST attribute id ID #IMPLIED>

<!ATTLIST name id ID #IMPLIED>

<!ATTLIST required id ID #IMPLIED>

<!ATTLIST rtexprvalue id ID #IMPLIED>

<!ATTLIST param-name id ID #IMPLIED>

<!ATTLIST param-value id ID #IMPLIED>

<!ATTLIST listener id ID #IMPLIED>

<!ATTLIST listener-class id ID #IMPLIED>
```

JSP.C.3DTD for TLD, JSP 1.1

The following is a DTD describing a Tag Library Descriptor file in JSP 1.1 format. This is the same DTD as http://java.sun.com/dtd/web-jsptaglibrary_1_1.dtd, except for some formatting changes to extract comments and make them more readable:

```
<!--
This is the DTD defining the JavaServer Pages 1.1 Tag Library descriptor (.tld)
(XML) file format/syntax.

A Tag Library is a JAR file containing a valid instance of a Tag Library Descriptor
(taglib.tld) file in the META-INF subdirectory, along with the appropriate imple-
menting classes, and other resources required toimplement the tags defined
therein.

Use is subject to license terms.
-->

<!--
The taglib tag is the document root, it defines:
tlibversion    the version of the tag library implementation
jspversion     the version of JSP the tag library depends upon
shortname      a simple default short name that could be used by a JSP authoring
tool to create names with a mnemonic value; for example, the it may be used as
the prefered prefix value in taglib directives
uri            a uri uniquely identifying this taglib
info           a simple string describing the “use” of this taglib, should be user dis-
cernable
-->

<!ELEMENT taglib (tlibversion, jspversion?, shortname, uri?, info?, tag+) >

<!ATTLIST taglib id ID #IMPLIED
      xmlns CDATA #FIXED
     “http://java.sun.com/dtd/web-jsptaglibrary_1_1.dtd”
>

<!--
Describes this version (number) of the taglibrary (dewey decimal)

#PCDATA ::= [0-9]*{ “.”[0-9] }0..3
-->

<!ELEMENT tlibversion (#PCDATA) >
```

```
<!--
Describes the JSP version (number) this taglibrary requires in order to function
(dewey decimal)

The default is 1.1

#PCDATA ::= [0-9]*{ “.”[0-9] }0..3
-->

<!ELEMENT jspversion  (#PCDATA) >

<!--
Defines a short (default) shortname to be used for tags and variable names used/
created by this tag library.  Do not use white space, and do not start with digits or
underscore.

#PCDATA ::= NMTOKEN
-->

<!ELEMENT shortname      (#PCDATA) >

<!--
Defines a public URI that uniquely identifies this version of the taglibrary Leave it
empty if it does not apply.
-->

<!ELEMENT uri (#PCDATA) >

<!--
Defines an arbitrary text string descirbing the tag library
-->

<!ELEMENT info(#PCDATA) >

<!--
The tag defines a unique tag in this tag library, defining:
- the unique tag/element name
- the subclass of javax.servlet.jsp.tagext.Tag implementation class
- an optional subclass of javax.servlet.jsp.tagext.TagExtraInfo
- the body content type (hint)
- optional tag-specific information
- any attributes
-->

<!ELEMENT tag (name, tagclass, teiclass?, bodycontent?, info?, attribute*) >
```

```
<!--
Defines the subclass of javax.serlvet.jsp.tagext.Tag that implements the request
time semantics for this tag. (required)

#PCDATA ::= fully qualified Java class name
-->

<!ELEMENT tagclass (#PCDATA) >

<!--
Defines the subclass of javax.servlet.jsp.tagext.TagExtraInfo for this tag. (option-
al)

If this is not given, the class is not consulted at translation time.

#PCDATA ::= fully qualified Java class name
-->

<!ELEMENT teiclass (#PCDATA) >

<!--
Provides a hint as to the content of the body of this tag. Primarily intended for use
by page composition tools.

There are currently three values specified:
tagdependent    The body of the tag is interpreted by the tag implementation it-
self, and is most likely in a different “langage”, e.g embedded SQL statements.
JSP             The body of the tag contains nested JSP syntax
empty           The body must be empty. The default (if not defined) is JSP

#PCDATA ::=  tagdependent | JSP | empty
-->

<!ELEMENT bodycontent (#PCDATA) >

<!--
The attribute tag defines an attribute for the nesting tag

An attribute definition is composed of:
- the attributes name (required)
- if the attribute is required or optional (optional)
- if the attributes value may be dynamically calculated at runtime by a scriptlet ex-
pression (optional)
-->
```

```
<!ELEMENT attribute (name, required? , rtexprvalue?) >

<!--
Defines the canonical name of a tag or attribute being defined

#PCDATA ::= NMTOKEN
-->

<!ELEMENT name(#PCDATA) >

<!--
Defines if the nesting attribute is required or optional.

#PCDATA ::= true | false | yes | no

If not present then the default is “false”, i.e the attribute is optional.
-->

<!ELEMENT required    (#PCDATA) >

<!--
Defines if the nesting attribute can have scriptlet expressions as a value, i.e the
value of the attribute may be dynamically calculated at request time, as opposed
to a static value determined at translation time.

#PCDATA ::= true | false | yes | no

If not present then the default is “false”, i.e the attribute has a static value
-->

<!ELEMENT rtexprvalue (#PCDATA) >

<!ATTLIST tlibversion id ID #IMPLIED>

<!ATTLIST jspversion id ID #IMPLIED>

<!ATTLIST shortname id ID #IMPLIED>

<!ATTLIST uri id ID #IMPLIED>

<!ATTLIST info id ID #IMPLIED>

<!ATTLIST tag id ID #IMPLIED>

<!ATTLIST tagclass id ID #IMPLIED>
```

```
<!ATTLIST teiclass id ID #IMPLIED>

<!ATTLIST bodycontent id ID #IMPLIED>

<!ATTLIST attribute id ID #IMPLIED>

<!ATTLIST name id ID #IMPLIED>

<!ATTLIST required id ID #IMPLIED>

<!ATTLIST rtexprvalue id ID #IMPLIED>
```

APPENDIX JSP.D

Page Encoding Detection

This appendix details the algorithm containers are required to use in order to determine the character encoding for a JSP file. See Chapter JSP.4, “Internationalization Issues” for details on where this algorithm is used. The algorithm is designed to maximize convenience to the page author, while preserving backwards compatibility with previous versions of the JSP specification.

JSP.D.1Detection Algorithm

The following is a complete though unoptimized algorithm for determining the character encoding for a JSP file. JSP containers may use an optimized version of this algorithm, but it must detect the same encoding as the algorithm in all cases.

1. Decide whether the source file is a JSP page in standard syntax or a JSP document in XML syntax.

    a. If there is a <is-xml> element in a <jsp-property-group> that names this file, then if it has the value "true", the file is a JSP document, and if it has the value "false", the file is not a JSP document.

    b. Otherwise, if the file name has the extension "jspx", the file is a JSP document.

    c. Otherwise, try to find a <jsp:root> element in the file.

        i. Determine the initial encoding from the first four bytes of the file, as described in appendix F.1 of the XML 1.0 specification. For the byte sequence "3C 3F 78 6D", use ISO-8859-1; for the byte sequence "4C 6F A7 94", use IBM037; for all other cases, use the UTF-* or UCS-* encoding given in the appendix.

ii. Read the file using the initial encoding and search for a <jsp:root> element. If the element is found and is the top element, the file is a JSP document in XML syntax

d. Otherwise, the file is a JSP page in standard syntax.

2. Reset the file.

3. If the file is a JSP page in standard syntax, use these steps.

   a. Check whether there is a JSP configuration element <page-encoding> whose URL pattern matches this file.

   b. Read the file using the initial encoding and search for a pageEncoding attribute in a page declaration. The specification requires the attribute to be found only if it is not preceded by non-ASCII characters, so simplified implementations are allowed.

   c. Report an error if there are a <page-encoding> configuration element whose URL pattern matches this file and a pageEncoding attribute, and the two name different encodings.

   d. If there is a <page-encoding> configuration element whose URL pattern matches this file, the page character encoding is the one named in this element.

   e. Otherwise, if there is a pageEncoding attribute, the page character encoding is the one named in this attribute.

   f. Otherwise, read the file using the initial encoding and search for a charset value within a contentType attribute in a page declaration. If it exists, the page character encoding is the one named in this charset value. The specification requires the attribute to be found only if it is not preceded by non-ASCII characters, so simplified implementations are allowed.

   g. Otherwise, the page character encoding is ISO-8859-1.

4. If the file is a JSP document in XML syntax, use these steps.

   a. Determine the page character encoding as described in appendix F.1 of the XML 1.0 specification. Note whether the encoding was named in the encoding attribute of the XML prolog or just derived from the initial bytes.

   b. Check whether there is a JSP configuration element <page-encoding> whose URL pattern matches this file.

c. Read the file using the detected encoding and search for a pageEncoding attribute in a <jsp:directive.page> element.

d. Report an error if any of the following conditions is met:

i. The XML prolog names an encoding and there is <page-encoding> configuration element whose URL pattern matches this file and which names a different encoding.

ii. The XML prolog names an encoding and there is a pageEncoding attribute which names a different encoding.

iii. There are a <page-encoding> configuration element whose URL pattern matches this file and a pageEncoding attribute, and the two name different encodings.

5. Reset the file and read it using the page character encoding.

APPENDIX JSP.E Changes

This appendix lists the changes in the JavaServer Pages specification. This appendix is non-normative.

JSP.E.1Changes between JSP 2.0 PFD3 and JSP 2.0 Final

- Minor typos and clarifications.
- API Changes:
  - Changed javax.servlet.jsp.tagext.JspFragment from an interface to an abstract class. Made JspFragment.invoke() abstract.
  - Added JspFragment.getJspContext() method.
- Added section on compatibility and porting issues between JSP 1.2 and JSP 2.0 to Preface.
- Minor clarifications to JSR-45 line number mapping guidelines.
- Clarified use of <jsp:output> in tag files.
- Added doctype-root-element, doctype-public and doctype-system properties to <jsp:output> for outputting DOCTYPE in JSP XML sytnax.
- Requires that the JSP stratum is the default, for JSR-45 debugging.
- Added I18N detection algorithm appendix.
- Added element structure diagrams for TLD schema.
- Removed requirement on ordering of attribute setter calls, except for <jsp:attribute>.

- Clarified that a TLD is invalid if it specifies "JSP" as the <body-content> for a SimpleTag extension.
- Made the JSR-45 requirement optional.
- Clarified ranges of EL integer and floating point literals.
- Clarified semantics for cross-syntax translation-time includes (between standard and XML syntaxes). Added three examples to illustrate these semantics.
- Loosened checking for duplicate page directive attributes and duplicate taglib directive declarations to make static includes more useful. Duplicates are now okay so long as the values are identical in both places.
- Re-enabled preludes and codas for JSP Documents (XML syntax).
- Removed special behavior of the id attribute for custom tags. Virtually no containers implement this feature and it was thought solidifying this requirement in JSP 2.0 would berak applications.
- Clarified that the uri passed to TagLibraryValidator.validate() is the uri in the XML View, not necessarily the value of the uri attribute in the taglib directive.

JSP.E.2Changes between JSP 2.0 PFD2 and JSP 2.0 PFD3

- Minor typos and clarifications.
- Added \$ as a way to quote $ in template text and attribute values, both in standard and XML syntaxes. This enabled quoting of EL expressions. Quoting of $ is disabled for pages where EL is ignored, for backwards compatibility. Described the XML view for quoting EL expressions.
- Changes to the API:
  - NullPointerException must be thrown for null name in various methods.
  - Allow null passed as default prefix in EL API to indicate a prefix is required.
  - SimpleTagSupport: Made jspBody and jspContext fields private. Made getJspBody() and getJspContext() accessors protected.
  - ExpressionEvaluator: Changed so that only one EL expression can be parsed or evaluated at a time, with no intermixed static text. Removed defaultPrefix parameters and changed so that FunctionMappers can mutate between ExpressionEvaluator.parseExpression() and Expression.evaluate().
  - Updated javadocs for JspWriter to indicate that the resulting text is written to the buffer or underlying writer directly, and not converted to the platform's

default encoding first, which would make no sense in this context.

- Changes to Tag Library Descriptor (TLD):
  - Added descriptionGroup, example and extension elements to <tag-file>.
  - Moved definitions of j2ee:extensibleType and j2ee:tld-extensionType to web-jsptaglibrary_2_0.xsd.
  - Added function-extension element.
  - Updated tag-name-uniqueness to check for uniqueness across name elements both in tag and tag-file elements. Removed tag-file-name-uniqueness.
  - Removed capital versions of TAGDEPENDENT, EMPTY, and SCRIPTLESS enumerations in body-contentType.
  - Reformatted indentation.
  - Added example of how to write a schema for a TLD extension.
- Changes to the Expression Language (EL):
  - Clarified that the container must check EL syntax at translation time.
  - Removed rules for escaping EL expression output. in EL chapter.
  - Added conditional operator (A ? B : C).
  - Added coercion rules for target type Long.
  - The empty operator can now be applied to any Collection.
  - In all cases, omitting the prefix of a function now means the function is associated with the default namespace.
- EBNF Grammar Changes:
  - Better handling for syntax errors for unmatched action tags
  - Added logic to handle quoting EL expressions.
- Changed conversion rules for attribute values for the empty String "" to match EL semantics.
- Removed synchronization of variables from the page to the tag file, but kept synchronization from tag file to page. This is consistent with classic tags.
- Changed the default value for the rtexprvalue attribute of the attribute directive to true.
- I18N Changes:
  - During a <jsp:forward> or <jsp:include> the container is now required to encode the parameters using the character encoding from the request object.

  - Character encoding is now determined for each file separately, even if one file includes another using the include directive.
- Changed the semantics of <is-xml> so that a value of false simply indicates the resource is not a JSP document, but rather a JSP page.
- Changed .jspx extension to only work with a Servlet 2.4 or greater web.xml.
- Synchronized behavior of error pages with the Servlet specification.
- Changed dynamic-attributes attribute of the tag directive to specify the name of a Map to place the dynamic attributes into, instead of placing them directly in the page scope. Dynamic attributes with a uri are ignored.
- Added alias attribute and name-from-attribute mechanism for tag files.
- Clarified behavior of Tag Library Validators when namespaces are redefined in JSP documents.
- Added non-normative guidelines for JSR-45 line number mapping.
- Clarified that DTD validation of JSP Documents must be done by containers.
- Clarified that in JSP Documents the prefix "jsp" is not fixed for the namespace http://java.sun.com/JSP/Page.
- Clarified that, if 'a' is not a custom action, <a href="%= url %"> does not contain a request-time attribute value whereas <a href="${url}"> does.

JSP.E.3Changes between JSP 2.0 PFD and JSP 2.0 PFD2

- Minor typos and clarifications.
- Clarified handling of non-String types when using <jsp:attribute>.
- Clarified that JSP Configuration settings do not apply to tag files.
- Changed the way EL expressions and Scripting is enabled/disabled:
  - Removed isScriptingEnabled attribute from page/tag directive.
  - Changed <scripting-enabled> JSP Configuration element to <scripting-invalid>
  - Changed <el-enabled> JSP Configuration element to <el-ignored>
  - Changed isELEnabled to isELIgnored.
- Clarified that EL expressions can be used to provide request-time attribute values as well.

- Added a grammar for the <function-signature> element in the TLD.
- Clarified expected container behavior for various illegal JSP code.
- Clarified JSP Configuration URL Patterns are as defined in the Servlet specification.
- Clarified that for <jsp:invoke>, an IllegalStateException must occur if scope is session and the calling page does not participate in a session.
- Clarified that invalid tag libraries must trigger a translation error.
- API Changes, including:
  - Various javadoc clarifications to enhance testability.
  - Added new pushBody(java.io.Writer) to JspContext.
  - Moved popBody() from PageContext to JspContext.
  - Removed ELException.toString()
  - Adjusted semantics of SimpleTagSupport.findAncestorWithClass() so that it uses the return value of TagAdapter.getAdaptee() when comparing class types, and for the final return value.
  - Clarified SkipPageException should not be manually thrown in JSP Pages.
  - Removed TagLibraryInfo.getTagdir() and corresponding protected attribute, as it can never return anything useful. Also removed the JSP 2.0 version of the constructor, since it only differed by its tagdir parameter.
  - Removed pContext parameter from VariableResolver.resolveVariable().
  - Changed ExpressionEvaluator from an interface to an abstract class.
  - Changed Expression from an interface to an abstract class.
  - Removed PAGE_SCOPE, REQUEST_SCOPE, SESSION_SCOPE and APPLICATION_SCOPE constants from JspContext as they are duplicated in PageContext.
- Various changes to schema for JSP portion of web.xml and to schema for TLDs.
- Made it illegal to refer to classes in the unnamed (a.k.a. default) package, since JDK 1.4 has stopped supporting this.
- Reduced J2SE requirement to J2SE 1.3 for standalone containers and J2SE 1.4 for J2EE 1.4 containers. Made Unicode 3.0 and JSR-45 optional when running in J2SE 1.3 and required when running in J2SE 1.4.

- JSR-45 SourceDebugAttribute extensions must now be generated for tag files as well.
- Internationalization Changes:
  - Renamed the "Localization" chapter to "Internationalization", and rewrote it for clarity, to provide more up-to-date information on JSTL, and to refer to the Servlet specification for details of the ServletResponse behavior.
  - If the contentType charset defaults to ISO-8859-1, it isn't passed on to the ServletResponse, so that implicit character encoding specifications can still override it in Servlet 2.4.
  - The page character encoding of documents in XML syntax is now always detected in the XML specification. The pageEncoding attribute and/or page-encoding configuration element may be given, but must not disagree with the XML prolog.
  - XML views are encoded in UTF-8, and their pageEncoding attribute is set to reflect this. Their contentType attribute is set to reflect the contentType that the container will pass to the ServletResponse.
- Moved details about XML view of tag files to "JSP and XML" chapter.
- Changed the way variable synchronization works in Tag Files and simple tag handlers:
  - Removed the Map parameter from JspFragment.invoke().
  - Removed all JspFragment logic dealing with preparing and restoring the page scope.
  - Disallowed the use of <jsp:param> in <jsp:invoke> and <jsp:doBody>
  - Removed fragment attribute from the variable directive (and from the variable element in the TLD). Variables can no longer be scoped to a specific fragment.
  - It is now a translation-time error to have a variable directive with a name-given attribute having the same value as the name attribute of an attribute directive, for a given tag file translation unit.
  - Variables appear as page-scoped attributes local to the tag file, and are synchronized with the calling JspContext at various points, depending on the scope of the variable.
  - Clarified that scripting variables are still declared for SimpleTag variables.
- Clarified what implicit objects are available to tag files.
- Removed the value attribute of the <jsp:body> standard action.

- Added glossary entries for tag file, JSP fragment, named attribute, JSP segment, classic tag handler, simple tag handler, dynamic attribute, and JSP configuration.
- Added <jsp:element> standard action to standard syntax as well.
- Expression Language
  - Clarified behavior of EL functions whose implementations are declared to return void.
  - Specified expected behavior when an EL function throws an exception.
  - Specified that the result of an EL expression in template text is quoted, to help in preventing cross-site scripting attacks.
  - Made rules for coercing A to Number type N more specific.
  - Added special handling for all operators for BigInteger and BigDecimal types.
- Specified stricter rules for tag handler instance reuse to improve compatibility.
- Changed behavior of JspException being thrown from dynamic attributes to be handled as though the exception came from another setter method, instead of having come from doStartTag() or doEndTag().
- Clarified how dynamic attributes behave with respect to namespaces.
- Relaxed the need to call setParent(null) on Simple Tag Handlers.
- Clarified that classic tag invocations with empty bodies will not cause body methods to be invoked, even if the body content type for the tag is something other than "empty".
- Some clarifications on how implicit taglib maps are constructed.
- EBNF Grammar Changes:
  - Fixed EBNF for Params, Fallback productions to allow for <jsp:body> to appear inside.
  - Clarified that <jsp:body> cannot be used to specify the body of <jsp:body> or <jsp:attribute> and that <jsp:attribute> cannot be used to specify an attribute of <jsp:attribute>.
  - Clarified that XML-style attributes, such as those used in directives, can be separated from each other by whitespace.
  - Added <jsp:output> and <jsp:text> to grammar.
  - Corrected definition of <jsp:param>.
  - Fixed EBNF for <jsp:text>, <jsp:scriptlet>, <jsp:expression> and <jsp:decla-

ration> so that CDATA sections are now allowed.

- Added mayscript attribute to <jsp:plugin> tag.
- Clarified <jsp-property-group> matching logic and how URL pattern overlaps are resolved between <jsp-property-group> and <servlet-mapping> elements.
- Clarified that a primitive cannot be used as the type of an attribute in tag files.
- Clarified that the default for the language attribute of the page directive is java.
- Moved <jsp:element> and <jsp:text> to Standard Actions chapter. Added <jsp:output> to Standard Actions chapter.
- Split XML chapter into two chapters - one on JSP Documents and one on XML Views of JSP Pages and JSP Documents. Rewrote large portions of JSP Documents chapter.
- Changed semantics of SimpleTag so that if a body is not present, setJspBody() is not called (instead of passing null).
- Added XML syntax for tag files (.tagx).
- Made preludes and codas illegal for JSP Documents (XML syntax).

JSP.E.4Changes between JSP 2.0 PFD1a and JSP 2.0 PFD

- Synchronized Standard Actions Chapter with JSP 1.2 Errata B.
- Clarifications in the Localization Chapter to make encoding table clearer.
- Changed TagAdapter to reduce confusion for container vendors.
- EL Function implementations no longer need to be in a non-abstract class.
- Updated BNF for EL to include functions.
- Removed the restriction that the value attribute of <jsp:body> cannot be used for Classic Tag Handlers.
- Various typographical edits and clarifications (scattered).
- In <jsp:doBody>, made it illegal to specify a <jsp:param> with the same name as a variable with a scope of AT_BEGIN or NESTED.
- Provided a brief, non-normative overview of the SimpleTag lifecycle in the SimpleTag javadocs for the convenience of developers.

- Added new include() method to PageContext, with flush parameter.
- Removed name attribute from tag directive.
- Changed semantics of tag file packaging, and modified XML Schema accordingly.
- Improved access to error information in error pages by adding javax.servlet.error.exception and providing access to other attributes via the EL.
- Filled in many missing javadoc entries in the JSP API.
- Clarified that, for tag files, if an optional attribute is declared but not passed in, no page-scoped variable is created (used to say value is null, which is illegal).
- Added TLD Deployment extensions to Tag Extensions chapter and TLD. These extensions are only for tool consumption.
- Cleaned up description of coercion rules in Expression Language chapter.
- Clarified that Dynamic Attributes must be considered to accept request-time expression values.
- Modified the concept of JSP documents. In JSP 1.2 we had two quite separate syntaxes and, for instance, elements like <jsp:expression> were not available in *classic* JSP syntax. In JSP 2.0 the same basic syntax is available everywhere, but a JSP page can be tagged as being an XML document and thus can benefit from XML concepts like well-formedness, validity, and entity definitions.
- Added configuration property <is-xml> to indicate that a JSP page is to be treated as an XML document (JSP Document).
- Incorporated new XML syntax details in the Core Syntax and in the JSP documents chapter. Renamed Chapter 6 from JSP Documents to JSP and XML.
- XML syntax versions of all JSP elements are now also available in JSP pages that are not JSP documents - this is a, backward compatible, extension from the JSP 1.2 semantics.
- Added default interpretation of extension ".jspx" to mean a JSP document (and thus, an XML document).
- Added a <jsp:element> element to dynamically generate XML elements.

- Clarified that when a <jsp:useBean> element is used in an scriptless page, or in an scriptless context (as in the body of an action so indicated), there are no Java scripting variables created but instead an EL variable is created.
- Clarified that EL expressions are available in all attributes of both standard and custom actions that accept run time expressions.
- Changed <jsp:invoke> and <jsp:doBody> to accept var attribute to store fragment output as a String. Changed semantics of varReader so that ${reader} no longer echoes the contents of the Reader and no longer resets the stream. The Reader can be passed to a custom action for further processing.
- Can now use <jsp:attribute> for any standard or custom action attribute, and can now use scriptlets and expressions in the body of <jsp:attribute> where it makes sense.
- Removed fragment-input directive and replaced with a new fragment attribute for the variable directive. Removed <fragment-attribute> and <fragment-input> elements in the TLD and replaced them with new <attribute> subelement called <fragment>, and new <variable> subelement called <fragment>. Updated APIs for tag info accordingly.
- Clarified that the implicit objects available to JSP pages under the EL are always available through the given names.
- The EL Evaluator API has continued to evolve. Among the changes there is now a FunctionMapper abstraction, and parsing errors are now reported through an ELParseException. The VariableResolver Interface now is obtained from the JspContext and abstracts its context.

JSP.E.5Changes between JSP 2.0 PD2 and JSP 2.0 PFD1a

- Removed restriction that containers must not reuse JspFragment instances.
- Added javax.servlet.jsp.tagext.JspTag to the API chapter.
- Fixed EBNF for Params, Fallback productions.
- Fixed some minor typos (scattered).
- Added uniqueness constraints to XML Schema for tag/name, tag-file/name and function/name.
- Added SkipPageException as an exception for indicating a page is to be skipped in JspFragments and Simple Tag Handlers. Replaces SKIP_PAGE

and EVAL_PAGE constants (only for Simple Tag Handlers - Classic Tag Handlers still use those constants).

- Clarified <jsp:attribute> can be used to specify only request-time expression attributes, and can be used for standard actions, and custom actions implemented using either Classic Tag Handlers or Simple Tag Handlers. Also clarified the <jsp:body> value attribute can only be used for Simple Tag Handlers and that <jsp:attribute> can be used to specify a fragment even for Classic Tag Handlers.
- Modified the page scope handling for Jsp Fragments and Tag Files to be much cleaner. Removed peekPageScope(), pushPageScope() and popPageScope(). Instead, fragments are assumed to share the page scope with its containing page, and tag files are required to create a Jsp Context Wrapper.
- Removed javax.servlet.jsp.tagext.AttributeNotSupportedException, and replaced it with a simple JspException which is just as effective.
- Added two constructors to JspTagException to allow specification of the root cause.
- Made jspContext and jspBody fields protected in SimpleTagSupport.

JSP.E.6Changes between JSP 2.0 PD1 and JSP 2.0 PD2

NOTE: JSP 2.0 PD2 was not released publicly.

- Updated I18N chapter to indicate Unicode 3.0 support and new details URL.
- Now requires JSR-45 strata name to be JSP.
- Clarified trim attribute of <jsp:attribute> is to be used at translation time.
- Fixed some minor typos (scattered).
- Renamed <el-evaluation> web.xml element to <el-enabled>
- Reorganized new features. Created a cohesive chapter about Tag Files. Simple Tag Handler details were moved to Tag Extensions and to the API chapter. Standard Action description was moved to Standard Action chapter.
- Added a root interface JspTag to cover Tag and SimpleTag.
- Moved all TLD DTDs to a single "Tag Library Descriptor Schemas" Appendix and added the new JSP 2.0 XML Schema to that appendix.

- Added JSP 2.0 XML Schema, which is imported by the Servlet 2.4 Web Application Deployment Descriptor.
- Updated page directive table and grammar to include isScriptingEnabled and isELEnabled.
- Added language, import, isScriptingEnabled and isELEnabled attributes to tag directive.
- Applied fixes to EBNF grammar based on JSP 2.0 Preview EA1 experience
- Clarified that jsp:id is now required and added TagExtraInfo.validate() and requirement that container call it instead of TagExtraInfo.isValid().
- Reorganized slightly the EL chapter to emphasize the parts of the language that do not depend on JSP details. Also removed the description of the API in that chapter: the javadoc-generated chapter is more complete.
- Function names now need to be unique within a tag library; arity is not used to disambiguate functions. This was done to simplify the EL language and the decision can be revisited in later releases based on usage experience.
- Some refinements to the EL API: a new method was added that accepts a VariableResolver instead of a JspContext, and the prefix/shortname map has been split into two separate maps.

JSP.E.7Changes between JSP 2.0 CD2 and JSP 2.0 PD1

- Moved all the JSP configuration description into its own chapter.
- Reordered the EBNF description to be at the end of JSP 1.3.
- Restored some pieces in the Syntax chapter that were lost in an editing operation. The only substantive piece was the description of the <include-prelude> and <include-coda> elements, which are now in the JSP configuration chapter.
- Added details on how to implement functions in EL.

JSP.E.8Changes between JSP 2.0 CD1 and JSP 2.0 CD2

E.8.1 Between CD2c and CD2

- Upgraded major version from JSP 1.3 to JSP 2.0, added section to the Preface explaining change.
- Added directive examples to JSP Fragments chapter.
- Moved section describing passing attribute values via <jsp:attribute> and <jsp:body> to syntax chapter and moved definitions of these two standard actions to Standard Actions chapter, from JSP Fragments chapter.
- Added optional scope attribute to <jsp:invoke> and <jsp:doBody>.
- Improved and simplified the way tag files are packaged. One can now package tag files in JARs or place them in a subdirectory of /WEB-INF/tags/ and access them without specifying a uri.
- Changed SimpleTag to not extend Tag. Added TagAdapter to handle tag collaboration, and removed dependency on PageContext in SimpleTag. These changes help make SimpleTag usable in environments other than Servlet request/response.
- Changed fragment invocation via <jsp:invoke> and <jsp:doBody> to be able to expose their result as a java.io.Reader object instead of a String. This is expected to be more efficient.
- Added <include-prelude> and <include-coda> elements to <jsp-property-group>. Added a description in the Syntax Chapter.
- Added a getExpressionEvaluator() method to JspContext (and, thus, to PageContext).
- Added better description of JSP configuration information to different chapters.
- Added to-do notes on EL to Syntax chapter, sketching where the information will go.
- Renamed elEvaluation property of page directive. The new name is isELEnabled, to be consistent with other properties.

E.8.2 Between CD2b and CD2c

- Fixed syntax table so that flush is optional in <jsp:include> standard action.
- Integrated EL grammar with JSP EBNF.
- Clarified doEndTag() description when SKIP_PAGE is returned.
- Added dynamic-attributes element in tag directive to describe a tag file that accepts dynamic attributes.
- Added SimpleTag, JspFragment, DynamicAttributes, AttributeNotSupportedException, ExpressionEvaluator, and VariableResolver classes to API. Added new API chapter for javax.servlet.jsp.el package.
- Added isScriptingEnabled directive and scripting-enabled JSP configuration element.
- Renamed jsp-group JSP configuration element to jsp-properties-group. Clarified conflict resolution rules.
- Clarified direction with EL function - details still to come.
- Added a chapter for EL API.
- Added description of page-encoding JSP configuration element to Localization chapter.

E.8.3 Between CD2a and CD2b

- Reordered "Users of JSP Technology" and "Basic Concepts" in the Overview section.
- Added <jsp-config> element to web.xml as a parent element for <taglib>. Added <jsp-group> as a new subelement to describe properties for a group of JSP pages that are described using <url-pattern> and other elements. Currently the only other element is <el-evaluation>, which can be used to describe whether EL evaluation is active or not by default.
- Modified the default rules for EL evaluation. Now, EL evaluation is always off, but it is very easy to add evaluation on through a <jsp-group> element.
- Various EBNF fixes
- Fixed some typos in Example Scenario in JSP_Fragments chapter
- Clarified issues on <jsp:forward> from within a tag file?

- Clarified issues on <jsp:attribute> and whitespace

E.8.4 Changes between CD1 and CD2a

- Added a part structure to the specification description. This helps provide guideance to the readers.
- Added a mechanism to pass attributes whose names are not known until runtime to tag handlers (Dynamic Attributes).
- Added getPageContext() to SimpleTag.
- Adjustment to i18n table to make defaultInputEncoding the default output encoding if unspecified.
- Moved EBNF description from Fragments chapter to Core Syntax.
- Improved EBNF description of <jsp:attribute> and <jsp:body>. Also, easier to read valid standard action attribute sets.

JSP.E.9Changes between JSP 2.0 ED1 and JSP 2.0 CD1

This is the first Community Draft of the JSP 2.0 specification.

E.9.5 JSP Fragments, .tag Files, and Simple Tag Handlers

- A new chapter on JSP fragments and supporting technologies such as the .tag mechanism and simple tag handlers:
  - JSP fragments allow a portion of JSP code to be encapsulated into a Java object which can be passed around and evaluated zero or more times.
  - The .tag mechanism allows page authors to use JSP syntax to write Custom Actions.
  - Simple tag handlers integrates tightly with JSP fragments and allows for a much easier and more natural invocation protocol for tag extensions.

E.9.6 Expression Language Added

- Added the Expression Language chapter, equivalent to that released in the JSP Standard Tag Library (JSTL) Public Draft, Appendix A.

- Updated the Expression Language chapter, including preliminary information on the API to invoke the EL evaluator.

E.9.7 EBNF Fixes

Various fixes to the EBNF, to handle CustomAction translation errors correctly. Improved readability by adding ATTR[] construct, to allow easier expression of XML-style attributes that can appear in any order.

E.9.8 I18N Clarifications

Incorporated JSP 1.2 errata_a. Clarified when container can call setContentType() and how it is possible to dynamically affect content type and character encoding from within a page or custom action.

E.9.9 Other Changes

- Updated Status, Preface, Changes chapters.
- Made support for jsp:id mandatory.
- Various typographical fixes.

JSP.E.10Changes Between JSP 1.2 Final Draft and JSP 2.0 ED1

This is the first expert draft of the JSP 2.0 specification.

E.10.10 Typographical Fixes and Version Numbers

Various typographical fixes that do not change any specification requirements, and version number updates for JSP 2.0. Various things were fixed from JSP 1.2 such as missing page numbers, repeated table numbers, etc.

E.10.11 Added EBNF Grammar for JSP Standard Syntax

A new section was added to the Syntax Chapter that presents a simple EBNF grammar for the standard (i.e. non-XML) JSP syntax. The grammar is intended to provide a concise syntax overview and to resolve any syntax ambiguities present in the specification.

E.10.12 Added Users of JavaServer Pages Section

A new section was added to the Overview Chapter that describes the various classes of users that make use of JSP technology, describing their role, the technology they're familiar with, and the sections of this specifications that are relevant to them.

E.10.13 Added Placeholders for Expression Language and Custom Actions Using JSP

Two new chapters were added in anticipation of the new Expression Language and Custom Actions Using JSP features.

E.10.14 Added Requirement for Debugging Support

A new section was added to the JSP Container Chapter requiring support for JSR-045 ("Debugging Support for Other Languages"). The precompilation protocol was also updated.

JSP.E.11 Changes Between PFD 2 and Final Draft

This is the final version approved by JCP Executive Comittee; the document was updated to reflect that status. All change bars were reset.

E.11.15 Added jsp:id mechanism

A new mechanism was added to allow willing JSP containers to provide improved translation-time error information from TagLibraryValidator classes. The signature of TagLibraryValidator.validate() was modified slightly, and a new ValidationMessage class was added. These objects act through a new attribute, jsp:id, which is optionally supported by a JSP container and exposed only through the XML view of a JSP page. Chapter JSP.10, Chapter JSP.7 (Section JSP.7.4.1.2) and Chapter JSP.13 (Section JSP.13.9.6) were affected.

E.11.16 Other Small Changes

- Made height & width be rtexprs. Section JSP.5.7 was affected.
- Added attribute value conversion from String literal to short and Short, and corrected conversion for char and Character in Table JSP.1-11.

- Corrected a statement on the allowed return values for doStartTag() for Tag, IterationTag and BodyTag.. PFD2 incorrectly indicated that "emtpy" tags could only return SKIP_BODY; the correct statement is that tags whose body-content is "empty" can only return SKIP_BODY.

E.11.17 Clarification of role of id

The mandated interpretations of the "id" attribute in Section JSP 2.13.3 (that id represents page-wide unique ids) and the "scope" attribute in Section JSP 2.13.4 (regarding the scope for the introduced variable) were not enforced by most (perhaps all?) containers, and were inconsistent with prevalent practices in custom tag library development. Essentially these sections were being interpreted as localized statements about the jsp:useBean standard action. This has been made explicit and the sections were moved to Chapter 5 to reflect that.

Sections JSP.2.13.3 and JSP.2.13.4, and Chapter 4 were affected.

E.11.18 Clarifications on Multiple Requests and Threading

- Clarify that TLV instances need be thread safe. This affected Section JSP.13.9.6.
- Clarify that a tag handler instance is actively processing only one request at a time; this happens naturally if the tag handler is instantiated afresh through new() invocations, but it requires spelling once tag handler pooling is introduced. This clarification affected Chapter JSP.13.

E.11.19 Clarifications on JSP Documents

Several clarifications in Chapter JSP.6.

- Reafirmed that, in a JSP page in XML syntax, the URI for jsp core actions is important, not the prefix.
- Clarify that <?xml ... ?> is not required (as indicated by the XML spec).
- Clarified further the interpretation of whitespace on JSP documents.

E.11.20 Clarifications on Well Known Tag Libraries

Clarified that a tag library author may indicate, through the description comment, that a tag handler may expose at runtime only some subset of the information described through the tag handler implementation class. This is useful

for specialized implementations of well-known tag libraries like the JSP standard tag library. This clarification affected the description of the tag element in Section JSP.7.3 and the description of Tag.setParent() and TagSupport.findAncestorWithClass().

Removed the last paragraph on Section JSP.7.3.9; we don't have any plans to remove the well-know URI mechanism.

In general cleaned up the presentation of the computation of the taglib map between a URI and a TLD resource path; the previous version was clunky.

E.11.21 Clarified Impact of Blocks

Clarified further the legal uses and the role of block constructs within scriptlets and nested actions. This affected small portions of Sections JSP.1.3.3, JSP.9.4, JSP.9.4.4 and JSP.13.9.10.

E.11.22 Other Small Clarifications

- Reafirmed more explicitly that the location of icons is relative to TLD file. Section JSP.7.3 was affected.
- Removed non-normative comment about JSR-045 in Section JSP.1.1.10.
- Removed the comment on errorPages needing to be JSP pages, they can also be static objects. This affects Table JSP.1-8.
- Reaffirmed that event listeners in a tag library are registered before the application is started. This affects Section JSP.7.1.9.
- Clarify when the use of quoting conventions is required for attribute values. Clarified that request-time attribute values follow the same rules. This affects Section JSP.1.3.5, Section JSP.1.6 and Section JSP.1.14.1.
- Clarified the interpretation of relative specifications for include directives and jsp:include and jsp:forward actions. This affected Section JSP.1.2.1, Section JSP.1.10.5, Section JSP.5.4 and Section JSP.5.5
- Corrected the inconsistency on the precompilation protocol in Section JSP.11.4.2 regarding whether the requests are delivered to the page or not; they are not.
- Clarified that the <type> subelement of <attribute> in the TLD file should match that of the underlying JavaBean component property.

- Spelled out the use of ClassLoader.getResource() to get at data from a TagLibraryValidator class.

JSP.E.12Changes Between 1.2 PFD 1b and PFD 2

Change bars are used in almost all chapters to indicate changes between PFD 1b and PFD 2. The exception are Chapters 12 and 13 which are generated automatically from the Java sources and have no change bars. Most changes are semantical, but some of them are editorial.

E.12.23 Added elements to Tag Library Descriptor

The Tag Library Descriptor (TLD) was extended with descriptive information that is useful to users of the tag library. In particular, a TLD can now be massaged directly (e.g. using an XSLT stylesheet) into an end-user document.

A new <example> element was added, as an optional subelement of <tag>. The existing <description> element was made a valid optional subelement of <variable>, <attribute> and <validator>.

Section JSP.7.3 and Appendix JSP.B were affected. The TLD 1.2 DTD and Schemas were also affected.

E.12.24 Changed the way version information is encoded into TLD

The mechanism used to provide version information on the TLD was changed. In the PFD the version was encoded into the namespace. In PFD2 the namespace is not intended to change unless there are non-compatible changes, and the version is encoded into the <jsp-version> element, which is now mandatory. The new URI for the namespace is "http://java.sun.com/JSP/TagLibraryDescriptor".

Chapter JSP.7 and Appendix JSP.B were affected.

E.12.25 Assigning String literals to Object attributes

It is now possible to assign string literals to an attribute that is defined as having type Object, as well as to a property of type Object. The valid type conversions are now all described in Section JSP.1.14.2, and used by reference in the semantics of <jsp:setProperty>.

E.12.26 Clarification on valid names for prefix, action and attributes

We clarified the valid names for prefixes used in taglib directives, element names used in actions, and attribute names.

E.12.27 Clarification of details of empty actions

The JSP 1.1 specification distinguishes empty from non-empty actions, although the description could be better. Unfortunately, the JSP 1.2 PFD1 draft did not improve the description. This draft improves the description by making it clear what methods are invoked when.

Chapters 1, 7 and 13 were affected.

E.12.28 Corrections related to XML syntax

We clarified several issues related to the XML syntax for JSP pages and to the XML view of a JSP page. Most changes are in Chapter JSP.6.

- Removed an inexistant flush attribute in the include directive at Chapter JSP.6.
- Changed the name of jsp:cdata to jsp:text, since its semantics are very similar to the text element in XSLT.
- Changed the way the version information is encoded into the XML syntax; the URI used now is not version-specific and instead there is a required version attribute of jsp:root.
- Clarified that JSP comments in a JSP page in JSP syntax are not preserved on the XML view of the page.
- Clarified that JSP pages in XML syntax should have no DOCTYPE.
- Clarified the treatment of include directives in the XML view of a JSP page.
- Clarified the format of the URIs to use in xmlns attributes for taglib directives, and corrected Appendix JSP.B.

E.12.29 Other changes

We clarified several other inconsistencies or mistakes

- Explicitly indicated which attributes are reserved (Section JSP.1.3.5) and which prefixes are reserved (Section JSP.1.10.2).

- Add a comment to the DTD for the TLD indicating that a DOCTYPE is needed and what its value is. No changes to the value.
- Removed the paragraph at the end of Section JSP.7.3.9 that used to contain non-normative comments on the future of "well kwown URIs".
- Corrected the description of the valid values that can be passed to the flush attribute of the include action in Section JSP.5.4.
- Clarified that <jsp:param> can only appear within <jsp:forward>, <jsp:include>, and <jsp:params>.
- Clarified that <jsp:params> and <jsp:fallback> can only appear within <jsp:plugin>.
- Resolved a conflict in Section JSP.5.4 between the Servlet and the JSP specification regarding how to treat modifications to headers in included actions.
- Section 10.1.1 in PFD1 incorrectly described the valid return values for doStartTag() in tag handlers that implement the BodyTag interface. The correct valid values are SKIP_BODY, EVAL_BODY_INCLUDE and EVAL_BODY_BUFFER. Section now indicates this.

JSP.E.13Changes Between 1.2 PFD and 1.2 PFD 1b

PFD 1b is a draft that has mostly formating and a few editorial changes. This draft is shown only to make it simpler to correlate changes between later drafts and the previous drafts.

Change bars are used to indicate changes between PFD 1 and PFD 1b.

JSP.E.14Changes Between 1.2 PD1 and 1.2 PFD

The following changes ocurred between the Public Draft 1 and the Proposed Final Draft versions of the JSP 1.2 specification.

E.14.30 Deletions

- Removed the resetCustomAttributes() method.

E.14.31 Additions

- Added constructors and methods to JspException to support a rootCause (paralleling the ServletException).
- Added a PageContext.handleException(Throwable) method.
- Added references to JSR-045 regarding debugging support.
- Added new TryCatchFinally interface to provide better control over exceptions in tag handlers.
- Added an implicit URI to TLD map for packaged tag libraries. This also provides support for multiple TLDs inside a single JAR file.
- Added pageEncoding attribute to page directive.
- Added material to Chapter JSP.4.
- Added TagValidatorInfo class.
- Added Section JSP.1.1.9 with a suggestion on extension convention for top and included JSP files.

E.14.32 Clarifications

- A tag handler object can be created with a simple “new()”; it needs not be a fully fledged Beans, supporting the complete behavior of the java.beans.Beans.instantiate() method.
- Removed the “recommendation” that the <uri> element in a TLD be a URL to anything.
- Clarified that extension dependency information in packaged tag libraries should be honored.
- Clarified invocation and lifecycle of TagLibraryValidator.
- Clarified where TLDs may appear in a packaged JAR file.
- Clarified when are response.getWriter().

E.14.33 Changes

- Moved a couple of chapters around
- Improved and clarified Chapter JSP.6.
- Moved the include directive back into Chapter JSP.1.
- Renamed javax.servlet.jsp.tagext.PageInfo to javax.servlet.jsp.tagext.PageData (for consistency with existing TagData).
- Added initialization parameters to TagLibraryInformation validation in TLD, adding a new <validator> element, renaming <validatorclass> to <validator-class> for consistency, and adding <init-param> as in the Servlet web.xml descriptor.
- Added method to pass the initialization parameters to the validator class and removed the use of TagLibraryInfo. Added prefix and uri String arguments to validate() method.
- Changed element names in TLD to consistently follow convention. New names are <tag-class>. <tei-class>, <tlib-version, <jsp-version>, <short-name> and <body-content>. <info> was renamed <description>.

JSP.E.15 Changes Between 1.1 and 1.2 PD1

The following changes ocurred between the JSP 1.1 and JSP 1.2 Public Draft 1.

E.15.34 Organizational Changes

- Chapter 8 and 10 are now generated automatically from the javadoc sources.
- Created a new document to allow longer descriptions of uses of the technology.
- Created a new I18N chapter to capture Servlet 2.3 implications and others (mostly empty for PD1).
- Removed Implementation Notes and Future appendices, as they have not been updated yet.

E.15.35 New Document

We created a new, non-normative document, “Using JSP Technology”. The document is still being updated to JSP 1.2 and Servlet 2.3. We moved to this document the following:

- Some of the non-normative Overview material.
- All of the appendix on tag library examples.
- Some of the material on the Tag Extensions chapter.

E.15.36 Additions to API

- jsp:include can now indicate “flush=’false’”.
- Made the XML view of a JSP page available for input, and for validation.
- PropertyEditor.setAsText() can now be used to convert from a literal string attribute value.
- New ValidatorClass and JspPage classes for validation against tag libraries.
- New IteratorTag interface to support iteration without BodyContent. Added two new constants (EVAL_BODY_BUFFERED and EVAL_BODY_AGAIN) to help document better how the tag protocol works; they are carefully designed so that old tag handlers will still work unchanged, but the old name for the constant EVAL_BODY_TAG is now deprecated.
- Added listener classes to the TLD.
- Added elements to the TLD to avoid having to write TagExtraInfo classes in the most common cases.
- Added a resetCustomAttributes() method to Tag interface.
- Added elements to the TLD for delivering icons and descriptions to use in authoring tools.

E.15.37 Clarifications

- Incorporated errata 1.1_a and (in progress) 1.1_b.

E.15.38 Changes

- JSP 1.2 is based on Servlet 2.3, in particular:
- JSP 1.2 is based on the Java 2 platform.

JSP.E.16Changes Between 1.0 and 1.1

The JSP 1.1 specification builds on the JSP 1.0 specification. The following changes ocurred between the JSP 1.0 final specification and the JSP 1.1 final specification.

E.16.39 Additions

- Added a portable tag extension mechanism with an XML-based Tag Library Descriptor, and a run-time stack of tag handlers. Tag handers are based on the JavaBeans component model. Adjusted the semantics of the uri attribute in taglib directives.
- Flush is now a mandatory attribute of jsp:include, and the only valid value is “true”.
- Added parameters to jsp:include and jsp:forward.
- Enabled the compilation of JSP pages into Servlet classes that can be transported from one JSP container to another. Added appendix with an example of this.
- Added a precompilation protocol.
- Added pushBody() and popBody() to PageContext.
- Added `JspException` and `JspTagException` classes.
- Consistent use of the JSP page, JSP container, and similar terms.
- Added a Glossary as Appendix JSP.F.
- Expanded Chapter 1 so as to cover 0.92’s "model 1" and "model 2".
- Clarified a number of JSP 1.0 details.

E.16.40 Changes

- Use Servlet 2.2 instead of Servlet 2.1 (as clarified in Appendix B), including distributable JSP pages.
- `jsp:plugin` no longer can be implemented by just sending the contents of `jsp:fallback` to the client.
- Reserved all request parameters starting with "jsp".

APPENDIX JSP.F

Glossary

This appendix is a glossary of the main concepts mentioned in this specification. This appendix is non-normative.

action An element in a JSP page that can act on implicit objects and other server-side objects or can define new scripting variables. Actions follow the XML syntax for elements with a start tag, a body and an end tag; if the body is empty it can also use the empty tag syntax. The tag must use a prefix.

action, standard An action that is defined in the JSP specification and is always available to a JSP file without being imported.

action, custom An action described in a portable manner by a tag library descriptor and a collection of Java classes and imported into a JSP page by a taglib directive.

Application Assembler A person that combines JSP pages, servlet classes, HTML content, tag libraries, and other Web content into a deployable Web application.

classic tag handler A tag handler that implements the javax.servlet.jsp.tagext.Tag interface.

component contract The contract between a component and its container, including life cycle management of the component and the APIs and protocols that the container must support.

Component Provider A vendor that provides a component either as Java classes or as JSP page source.

distributed container A JSP container that can run a Web application that is tagged as distributable and is spread across multiple Java virtual machines that might be running on different hosts.

declaration A scripting element that declares methods, variables, or both in a JSP page. Syntactically it is delimited by the <%! and %> characters.

directive An element in a JSP page that gives an instruction to the JSP container and is interpreted at translation time. Syntactically it is delimited by the <%@ and %> characters.

dynamic attribute An attribute, passed to a custom action, whose name is not explicitly declared in the tag library descriptor.

element A portion of a JSP page that is recognized by the JSP translator. An element can be a directive, an action, or a scripting element.

EL expression An element in a JSP page representing an expression to be parsed and evaluated via the JSP Expression Language. Syntactically it is delimited by the ${ and } characters.

expression Either a scripting expression or an EL expression.

fixed template data Any portions of a JSP file that are not described in the JSP specification, such as HTML tags, XML tags, and text. The template data is returned to the client in the response or is processed by a component.

implicit object A server-side object that is defined by the JSP container and is always available in a JSP file without being declared. The implicit objects are request, response, pageContext, session, application, out, config, page, and exception for scriptlets and scripting expressions. The implicit objects are pageContext, pageScope, requestScope, sessionScope, applicationScope, param, paramValues, header, headerValues, cookie and initParam for EL expressions**.**

JavaServer Pages technology An extensible Web technology that uses template data, custom elements, scripting languages, and server-side Java objects to return dynamic content to a client. Typically the template data is HTML or XML elements, and in many cases the client is a Web browser.

JSP container A system-level entity that provides life cycle management and runtime support for JSP and servlet components.

JSP configuration The deployment-time process by which the JSP container is declaratively configured using a deployment descriptor.

JSP file A text file that contains JSP elements, forming a complete JSP page or just a partial page that must be combined with other JSP files to form a complete page. Most top-level JSP files have a .jsp extension, but other extensions can be configured as well.

JSP fragment A portion of JSP code, translated into an implementation of the javax.servlet.jsp.JspFragment abstract class.

JSP page One or more JSP files that form a syntactically complete description for processing a request to create a response.

JSP page, frontA JSP page that receives an HTTP request directly from the client. It creates, updates, and/or accesses some server-side data and then forwards the request to a presentation JSP page.

JSP page, presentation A JSP page that is intended for presentation purposes only. It accesses and/or updates some server-side data and incorporates fixed template data to create content that is sent to the client.

JSP page implementation class The Java programming language class, a servlet, that is the runtime representation of a JSP page and which receives the request object and updates the response object. The page implementation class can use the services provided by the JSP container, including both the servlet and the JSP APIs.

JSP page implementation object The instance of the JSP page implementation class that receives the request object and updates the response object.

JSP segment A portion of JSP code defined in a separate file, and imported into a page using the include directive.

named attribute A standard or custom action attribute whose value is defined using the <jsp:attribute> standard action.

scripting element A declaration, scriptlet, or expression, whose tag syntax is defined by the JSP specification, and whose content is written according to the scripting language used in the JSP page. The JSP specification describes the syntax and semantics for the case where the language page attribute is java**.**

scripting expression A scripting element that contains a valid scripting language expression that is evaluated, converted to a String, and placed into the implicit out object. Syntactically it is delimited by the <%= and %> characters.

scriptlet An scripting element containing any code fragment that is valid in the scripting language used in the JSP page. The JSP specification describes what

is a valid scriptlet for the case where the language page attribute is java**.** Syntactically a scriptlet is delimited by the <% and %> characters.

simple tag handler A tag handler that implements the javax.servlet.jsp.tagext.SimpleTag interface.

tag A piece of text between a left angle bracket and a right angle bracket that has a name, can have attributes, and is part of an element in a JSP page. Tag names are known to the JSP translator, either because the name is part of the JSP specification (in the case of a standard action), or because it has been introduced using a Tag Library (in the case of custom action).

tag file A text-based document that uses fixed template data and JSP elements to define a custom action. The semantics of a tag file are realized at runtime by a tag handler.

tag handler A Java class that implements the JspTag interface and is the runtime representation of a custom action.

tag library A collection of custom actions described by a tag library descriptor and Java classes.

tag library descriptor An XML document describing a tag library.

Tag Library Provider A vendor that provides a tag library. Typical examples may be a JSP container vendor, a development group within a corporation, a component vendor, or a service vendor that wants to provide easier use of their services.

web application An application built for the Internet, an intranet, or an extranet.

web application, distributable A Web application that is written so that it can be deployed in a Web container distributed across multiple Java virtual machines running on the same host or different hosts. The deployment descriptor for such an application uses the distributable element.

Web Application Deployer A person who deploys a Web application in a Web container, specifying at least the root prefix for the Web application, and in a J2EE environment, the security and resource mappings.

web component A servlet class or JSP page that runs in a JSP container and provides services in response to requests.

Web Container Provider A vendor that provides a servlet and JSP container that support the corresponding component contracts.