JavaServer Pages™ Specification

Version 1.2 - public draft 1 (PD1)

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

August 15, 2000

Eduardo Pelegrí-Llopart, editor

901 San Antonio Road
Palo Alto, CA 94303 USA
650 960-1300 fax: 650 969-9131

JavaServer Pages(TM) (JSP) Specification ("Specification")

Version: 1.2
Status: Pre-FCS
Release: August 15, 2000

NOTICE

The Specification is protected by copyright and the information described therein may be protected by one or more U.S. patents, foreign patents, or pending applications. Except as provided under the following license, no part of the Specification may be reproduced in any form by any means without the prior written authorization of Sun Microsystems, Inc. ("Sun") and its licensors, if any. Any use of the Specification and the information described therein will be governed by the terms and conditions of this license and the Export Control and General Terms as set forth in Sun's website Legal Terms. By viewing, downloading or otherwise copying the Specification, you agree that you have read, understood, and will comply with all of the terms and conditions set forth herein.

Subject to the terms and conditions of this license, Sun hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense) under Sun's intellectual property rights to review the Specification internally for the purposes of evaluation only. Other than this limited license, you acquire no right, title or interest in or to the Specification or any other Sun intellectual property. The Specification contains the proprietary and confidential information of Sun and may only be used in accordance with the license terms set forth herein. This license will expire ninety (90) days from the date of Release listed above and will terminate immediately without notice from Sun if you fail to comply with any provision of this license. Upon termination, you must cease use of or destroy the Specification.

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, 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" AND IS EXPERIMENTAL AND MAY CONTAIN DEFECTS OR DEFICIENCIES WHICH CANNOT OR WILL NOT BE CORRECTED BY SUN. 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 CON-

TENTS 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 based on your use of the Specification for any purposes other than those of internal evaluation, and from 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

If this Software 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 Software 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 evaluation 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.

Contents

Status

The Java Community Process

This specification is being developed following the Java Community Process (JCP - see http://java.sun.com/jcp). 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 the comments back to the original expert group. 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 term *proposed final draft* is used in the JCP to indicate a version of the spec that is believed to be complete and ready but has not been validated by final test suites, implementation efforts and public feedback. The expert group may perform changes to the specification based on this feedback, but changes will be relatively minor.

When the expert group determines that it has a specification that meets its needs, and there is both a conformance test suite and a final reference implementation that implements the specification and passes the test suite, the expert group will submit the *final draft* for approval by the *Executive Comittee.*

It is important to emphasize that **any draft that is not final can change**, perhaps even in significant ways. Vendors, in particular, should use judgement in deciding what parts of the specification they should start implementing. The expert group will try to convey the confidence level of specific features as well as possible and will not indulge in gratituous changes.

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 JCP indicates minimum requirements regarding the availability of the reference implementation and test suites but individual specifications can have additional requirements. The reference implementation for JSP 1.2 and Servlet 2.3 will be developed as an open source project, under an agreement with the Apache Software Foundation.

This Draft

This document is the Public Draft 1 (PD1) of the JavaServer Pages 1.2 Specification. This draft is intended to include all features in JSP 1.2. The specification will change based on feedback from a number of sources including:

- Public.
- Implementation efforts.
- Test suite efforts.
- Use of the implementation to write actual applications.

This draft includes an up-to-date changes appendix.

We expect to release at least one more public draft before the end of this year.

We expect to submit the final spec for approval by the EC in the second quarter of the year 2001.

Errata Work

Most of JSP 1.2 specification is carried over from the JSP 1.1 specification. As vendors and users have been using the JSP 1.1 technology in earnest, issues have surfaced in the specification document. These issues are being collected as erratas and are being propagated into the JSP 1.2 specification. Erratas are being discussed in a forum that includes the large majority of the JSP 1.1 expert group, plus a significant number of other vendors, implementors and the like, under the facilitation of Eduardo Pelegri-Llopart in the role of the “JSP 1.1 interpretation guru”. This group, commonly refered to by the name of the mailing list it uses - *jsp-tfaq-comments@eng.sun.com* - operates without any non-disclosure agreement and has open membership. Contact *pelegri@eng.sun.com* if you feel you should be included.

The first errata (1.1_a) is now available for public review at the public JSP web site. A second errata (1.1_b) is the last planned errata and is being worked on.

Preface

This is the expert draft 3 of the *JavaServer Pages™ 1.2 Specification*. This specification is being developed following the Java Community Process. Comments from Experts, Participants, and the Public will be reviewed and incorporated into the specification where applicable.

JSP 1.2 extends JSP 1.1 in a number of ways, including:

- Using Servlet 2.3 as the foundations for its semantics.
- Correcting and making available the mapping from a JSP page to an XML document, and exploiting this in authoring tools and translation-time validation.
- Improving on authoring support.
- Improving on I18N support.
- Fixing the infamous “flush before you include” limitation in JSP 1.1.
- Refinements on tag library runtime support.

Unlike JSP 1.1, JSP 1.2 assumes the Java 2 platform.

Details on the conditions under which this document is distributed are described in the license on page 2.

Who should read this document

This document is intended to be the authoritative description of the JSP 1.2 specification. Although the specification is not intended to be overly formal, it is not a User’s Guide, and we expect other documents to be created that will cater to different readerships.

A companion document to this specification: “Using JavaServer Pages™ Technology” provides an overview of the technology and includes descriptions of different methodologies for using it. The “Using” document is intended to lag this specification only briefly and it is not going to be a replacement for the type of in-depth presentation and guideance that we expect will be found in other material that will follow.

Related Documents

JSP 1.2 assumes the Java 2 platform version 1.2.

Implementors of JSP containers and authors of JSP pages will be interested in a number of other documents, of which the following are worth mentioning explicitly.

TABLE 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/1.2
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

Historical Note

We would like to remember the original individuals that started the web server work in the Java platform. James Gosling started writing a Web Server in Java in 1994/1995, that became the foundation for Servlets. A larger project emerged in 1996 with Pavani Diwanji as lead engineer and many other key members. From this project came the Java Web Server product at Sun.

>>>MORE HISTORY STILL TO BE FILLED IN<<<

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 has seen 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 Enterprise Edition platform. There has also been increased activity on tag libraries and on many different approaches on how to organize all these features together. See the “Using JavaServer Pages™ Technology” for details on that area.

It is impossible to track the industry in a printed document; 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. We want to thank the community that implemented the reference implementation, and the vendors that have implemented the spec. >>**MORE NAMES HERE**<<.

We want to thank all the book authors, and the web sites that are tracking and facilitating the creation of the JSP community. >> **MORE NAMES HERE**<<

Last, but certainly not least important, 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.

CHAPTER 1

Overview

This chapter provides an overview of the JavaServer Pages technology.

1.1 The JavaServer Pages™ Technology

JavaServer Pages™ technology is the Java™ technology in the J2EE platform for building applications containing dynamic Web content such as HTML, DHTML, XHTML and XML. The JavaServer Pages technology enables the authoring of Web pages that create dynamic content easily but with maximum power and flexibility.

Basic Concepts

The JavaServer Pages technology provides a textual description for the creation of a *response* from a *request*. The technology builds on the following concepts:

- *Template Data*

  Substantial portions of most dynamic content is actually fixed. The JSP technology allow for the natural manipulation of this data.

- *Addition of Dynamic Data*

  The JSP technology allows the addition of dynamic data to the template data in a way that is simple yet powerful.

- *Encapsulation of Functionality*

  The JSP technology provides two related mechanisms for the encapsulation of functionality: the standard JavaBeans component architecture and the tag library mechanism.

- *Good Tool Support*

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

The result is a flexible and powerful server-side technology.

Benefits of the JavaServer Pages Technology

The JavaServer Pages technology offers a number of benefits:

- *Write Once, Run Anywhere™ properties*

  The JavaServer Pages technology is platform independent, both in its dynamic Web pages, its Web servers, and its underlying server components. You can author JSP pages on any platform, run them on any Web server or Web enabled application server, and access them from any Web browser. You can also build the server components on any platform and run them on any server.

- *High quality tool support*

  The Write Once, Run Anywhere properties of JSP allows the 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 support the separation of roles: *developers* write components that interact with server-side objects; *authors* put static data and dynamic content together to create presentations best suited for their intended audiences. Each of these roles emphasizes different types of abilities and, although these abilities may all be present in the same individual, they most commonly will not. A subset of the developer community may be focused in creating reusable components intented to be used by authors.

- *Reuse of components and tag libraries*

  The JavaServer Pages technology emphasizes the use of reusable components such as: JavaBeans™ components, Enterprise JavaBeans™ components and tag libraries. These components can be used in interactive tools for component development and page composition. This saves considerable development time while giving the cross-platform power and flexibility of the Java programming language and other scripting languages.

- *Separation of dynamic and static content*

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

- *Support for scripting and actions*

The JavaServer Pages technology supports scripting elements as well as actions. Actions permit the *encapsulation* of useful functionality in a convenient form that can also be manipulated by tools; scripts provide a mechanism to *glue together* this functionality in a per-page manner.

- *Web access layer for N-tier enterprise application architecture(s)*
  The JavaServer Pages technology is an integral part of the Java 2 Platform Enterprise Edition (J2EE), which brings Java technology to enterprise computing. You can now develop powerful middle-tier server applications, using a Web site that uses JavaServer Pages technology as a front end to Enterprise JavaBeans components in a J2EE compliant environment.

1.2 Basic Concepts

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

1.2.1 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 some dynamic actions and leverages on the Java 2 Platform.

The features in the JSP technology support a number of different paradigms for authoring of dynamic content; the document "*Using the JavaServer Pages(tm) Technology*" expands on this topic.

1.2.2 Web Applications

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

- 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.
- Client-side Java Applets, JavaBeans components, and arbitrary Java class files
- Java Runtime Environment(s) (downloadable via the Plugin) running in client(s)

JThe JavaServer Pages specification inherits from the Servlet specification the concepts of Applications, ServletContexts, Sessions, Requests and Responses. See the Java Servlet 2.3 specification for more details.

1.2.3 Components and Containers

JSP pages and Servlet classes are collectively referred 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 reuse of components, with quality-of-service features being provided by the container.

1.2.4 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 result of the translation phase is the creation of a Servlet class: the *JSP page implementation class* which will be instantiated at request time. The *JSP page implementation object* handles requests and creates responses.

It is possible to perform the translation phase early (what sometimes is called compiling the JSP pages into Servlets) and deliver in a Web Application, transparently, a Servlet class that will behave as the textual representation of the JSP page.

The translation phase may also be done by the JSP container at *deployment time*, or *on-demand* as the requests reach a JSP page that has not yet been translated.

1.2.5 Features in JSP

The key features of JavaServer Pages are:

- Standard directives
- Standard actions

- Scripting elements
- Tag Extension mechanism
- Template content

1.2.6 JSP Pages and the Java 2 Enterprise Edition Specification

Most of the integration of JSP pages within the J2EE 1.3 platform is inherited from the reliance on the Servlet 2.3 specification.

CHAPTER 2

Core Syntax and Semantics

This chapter describes the core syntax and semantics of the JavaServer Pages (JSP) 1.2 Specification.

2.1 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, possibly creating and/or using some other objects.

A JSP page describes this mapping by defining a *JSP page implementation class*, a subclass of Servlet (see Chapter 6) that implements the semantics of the JSP page. At request time, a request intended for a JSP page is delivered to a JSP page implementation object of the appropriate class.

All JSP containers must support HTTP as a protocol for requests and responses, but a container may also support additional request/response protocols. The default request and response objects (see XXXX) are of type HttpServletRequest and HttpServletResponse, respectively.

2.1.1 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 to that of a JSP container.

A Web component is either a Servlet or a JSP page. A web component may use the services of its container. The servlet element in a web.xml deployment descriptor is used to describe both types of web components; note that most JSP page components are defined implicitly in the deployment descriptor through the use of an implicit *.jsp* extension mapping.

2.1.2 XML Document for a JSP Page

All JSP pages have an equivalent XML document. This *equivalent* XML document is the view of the JSP page that is exposed to the translation phase (see below).

A JSP page can also be written directly as its equivalent XML document. Unlike in JSP 1.0 and JSP 1.1 containers, the XML document itself can be delivered to a JSP container for processing.

It is not valid to intermix “standard syntax” and XML syntax inside the same source file.

A JSP page (in either syntax) can include via a directive a JSP page in any syntax. I.e. within each unit one syntax is used but each unit can use either syntax.

2.1.3 Translation and Execution Phases

A JSP container is responsible for two separate activities. One is determining a JSP page implementation class that corresponds to a given JSP page. The other is managing 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. The process is determined by the semantics of the JSP pages, of the standard directives and actions, and of the custom actions in the tag libraries used in the page. A tag library can optionally provide a transformation to extend the translation phase, and a validation method to validate that a JSP page is correctly using the library.

A JSP container has some freedom in the details of the JSP page implementation class which it may exploit to address quality-of-service (most notably performance) issues.

During the *execution phase* the JSP container delivers events to the JSP page implemention object. The container is also responsible for instantiating request and response objects. The details of the contract between the JSP page implementation class and the JSP container is described in Chapter 6.

If the JSP page is delivered to the JSP container in source form, the translation of a JSP source page can occur at any time between initial deployment of the JSP page into the runtime environment of a JSP container and the receipt and processing of a client request for the target JSP page. Section 2.1.5 describes how to perform the translation phase ahead of deployment.

2.1.4 Events Exposed to JSP Pages

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

In JSP 1.1 only *init* and *destroy* events can be described: the first time a request is delivered to a JSP page a *jspInit()* method, if present, will be called to prepare the page. Similarly, a JSP container can reclaim the resources used by a JSP page at any time that a request is not being serviced by the JSP page by invoking first its *jspDestroy()* method; this is the same life-cycle as that of *Servlets.*

2.1.5 Compiling JSP Pages

JSP pages may be *compiled* into its JSP page implementation class plus some deployment information. This enables the use of JSP page authoring tools and JSP tag libraries to author a Servlet. This has several benefits:

- Removal of the start-up lag that occurs when a JSP page delivered as source receives the first request.
- Reduction of the footprint needed to run a JSP container, as the java compiler is not needed.

If a JSP page implementation class depends on some support classes in addition to the JSP 1.2 and Servlet 2.3 classes, the support classes will have to be included in the packaged WAR so it will be portable across all JSP containers.

A JSP page is compiled in the context of some Web Application, which provides resolution to relative URL specifications that are used in include directives (and elsewhere), taglib references, and translation-time actions used in custom actions.

A JSP page can also be compiled at deployment time.

Appendix A contains two examples of packaging of JSP pages. One shows a JSP page that is delivered in source form (probably the most common case) within a WAR. The other shows how a JSP page is translated into a JSP page implementation class plus deployment information indicating the classes needed and the mapping between the original URL that was directed to the JSP page and the location of the Servlet.

2.2 Web Applications

A Web Application is a collection of resources that are available through some URLs. A prototypical Web application can be composed from:

- 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.
- Client-side Java Applets, JavaBeans components, and arbitrary Java class files
- Java Runtime Environment(s) (downloadable via the Plugin) running in client(s)

Web applications are described in more detail in the Servlet 2.3 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 2.3 specification.

JSP 1.2 requires that all these resources are to be implicitly associated with and accessible through a unique ServletContext instance, which is available as the application implicit object (Section 2.8.3).

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 (Section 4.1.1)
- The jsp:include action element (Section 4.2.4).
- The jsp:forward action (Section 4.2.5).

JSP 1.2 supports portable packaging and deployment of Web Applications through the Servlet 2.3 specification. The JavaServer Pages specification inherits from the Servlet specification the concepts of Applications, ServletContexts, Sessions, Requests and Responses.

2.2.1 Relative URL Specifications within an Application

Elements may use *relative URL specifications*, which are called “URI paths” in the Servlet 2.1 specification. These paths are as in RFC 2396 specification; i.e. only the path part, no scheme nor authority. Some examples are:

```
“myErrorPage.jsp”
“/errorPages/SyntacticError.jsp”
“/templates/CopyrightTemplate.html”
```

When such a path starts with a “/”, it is to be interpreted by the application to which the JSP page belongs; i.e. its `ServletContext` object provides the base context URL. We call these paths “context-relative paths”.

When such a path does not start with a “/”, it is to be interpreted relative to the current JSP page: the current page is denoted by some path starting with “/” which is then modified by the new specification to produce a new path that starts with “/”; this final path is the one interpreted through the `ServletContext` object. We call these paths “page-relative paths”.

The JSP specification uniformly interprets all these paths in the context of the Web server where the JSP page is deployed; i.e. the specification goes through a map translation. The semantics applies to translation-time phase, and to request-time phase.

2.3 Syntactic Elements of a JSP Page

This section describes the basic syntax rules of the JSP pages.

2.3.1 Elements and Template Data

A JSP page has some *elements* and some *template data*. An element is an instance of an *element type* that are known to the JSP container; *template data* is everything else: i.e. 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 also 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.

2.3.2 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

The interpreation of an *action* may, and often will, depend on the details of the specific request received by the JSP page; actions provide information for the request processing phase. Actions can either be *standard*, i.e. defined in this specification, or *custom*, i.e. provided via the portable tag extension mechanism.

Action elements follow the syntax of XML elements: they have either a start tag (including the element name) possibly with attributes, an optional body, and a matching end tag, or they have an empty tag possibly with attributes:

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

and

```
<mytab attr1=”attribute value” .../>
```

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 some *objects* and may make them available to the scripting elements through some *scripting-specific variables.*

Scripting Elements

Scripting elements provide *glue* around template text and actions. There are three types of scripting elements: *declarations*, *scriptlets* and *expressions*. Declarations follow the syntax <%! ... %>; scriptlets follow the syntax <% .... %>; expressions follow the syntax <%= ... %>.

2.3.3 Start and End Tags

Elements that have distinct start and end tags (with enclosed body) must start and end in the same file. You cannot begin a tag in one file and end it in another.

This applies also 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.

2.3.4 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.

2.3.5 Attribute Values

Following the XML specification, attribute values always appear quoted. Both single and double quotes can be used. The entities ' and " are available to describe single and double quotes.

See also Section 2.13.1, “Request Time Attribute Values.

2.3.6 White Space

In HTML and XML, white space is usually not significant, with some exceptions. One exception is that an XML file must start with the characters <?xml, with no leading whitespace characters.

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

For example, since directives generate no data and apply globally to the JSP page, the following input file is translated into the corresponding result file:

For this input,

	`<?xml version=”1.0” ?>`
This is the default value	`<%@ page buffer=”8kb” %>`
	`The rest of the document goes here`

The result is

	`<?xml version=”1.0” ?>`
note the empty line	
	`The rest of the document goes here`

`As another example`, for this input,

	`<% response.setContentType(“....”);`
note no white between the two elements	`whatever... %><?xml version=”1.0” ?>`
	`<%@ page buffer=”8kb” %>`
	`The rest of the document goes here`

The result is

no leading space	`<?xml version=”1.0” ?>`
note the empty line	
	`The rest of the document goes here`

2.4 Error Handling

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

2.4.1 Translation Time Processing Errors

The translation of a JSP page source into a corresponding JSP page implementation class using the Java technology by a JSP container can occur at any time between initial deployment of the JSP page into the runtime environment of a JSP container, and the receipt and processing of a client request for the target JSP page. If translation occurs prior to the JSP container receiving a client request for the target (untranslated) JSP page then error processing and notification is implementation dependent. Fatal translation failures shall result in subsequent client requests for the translation target to also be failed with the appropriate error; for HTTP protocols, error status code 500 (Server Error).

2.4.2 Request Time Processing Errors

During the processing of client requests, arbitrary runtime 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. Such errors 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.

However, any uncaught exceptions thrown from 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 offending JSP page (or the implementation default behavior, if none is specified).

The offending `java.lang.Throwable` describing the error that occurred is stored in the `javax.ServletRequest` instance for the client request using the `putAttribute()` method, using the name "`javax.servlet.jsp.jspException`". Names starting with the prefixes "`java`" and "`javax`" are reserved by the different specifications of the Java platform; the "`javax.servlet`" prefix is used by the Servlet and JSP specifications.

If the `errorPage` attribute of a `page` directive names a URL that refers to another JSP, and that JSP indicates that it is an error page (by setting the `page` directive's `isErrorPage` attribute to `true`) then the "`exception`" implicit scripting language variable of that page is initialized to the offending `Throwable` reference

1. Note that this is independent of scripting language; this 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.

2.5 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.

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. If the generated comment is to have dynamic data, this can be obtained through an expression syntax, as in:

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

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 to “comment out” some portions of a JSP page. Note that JSP comments do not nest.

Note that an alternative way to place a “comment” in JSP is to do so by using the comment mechanism of the scripting language. For example:

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

2.6 Quoting and Escape Conventions

The following quoting conventions apply to JSP pages. Anything else is not processed.

Quoting in Scripting Elements

- A literal %> is quoted by %\>

Quoting in Template Text

- A literal <% is quoted by <\%

Quoting in Attributes

- A ‘ is quoted as \’
- A “ is quoted as \”
- A \ is quoted as \\
- A %> is quoted as %\>
- A <% is quoted as <\%

XML Representation

The quoting conventions are different to those of XML. See Chapter 5.

2.7 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 7. Most details are not language specific and are described in this chapter.

Most of the content of a JSP page is devoted to describing what data is written into the output stream of the response (usually sent back to the client). The description is based on a `JspWriter` object that is exposed through the implicit object *out* (see Section 2.8.3, “Implicit Objects). Its value varies:

- Initially, *out* is a new `JspWriter` object. This object may be different from the stream object from *response.getWriter(),* and may be considered to be interposed on the latter in order to implement buffering (see Section 2.10.1, “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`.

- Within the body of some actions, *out* may be temporarily re-assigned to a different (nested) instance of `JspWriter` object. Whether this is or is not the case depends on the details of the actions semantics. Typically the content, or the results of processing 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 that previous (nesting) stream. Such nested streams are always buffered, and require explicit flushing to a nesting stream or discarding of their contents.
- 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 1.1, the only events that can be described are initialization and destruction of the page; these are described using “well-known method names” in declaration elements (see page 81). Future specifications will likely define more events as well as a more structured mechanism for describing the actions to take.

2.8 Objects

A JSP page can access, create, and modify server-side objects. Objects can be made visible to actions and to scripting elements. Actions can access objects using a name in the PageContext object. Scripting elements can also have access to some objects directly via a *scripting variable*. Some *implicit objects* are visible via scripting variables in any JSP page.

A default file can be used to modify consistently the list of objects that are automatically visible in a page.

An object has a *scope* describing what entities can access the object.

When an object is exposed through a scripting variable the variable has a *scope* within the page.

2.8.1 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 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 are available directly in the JavaScript™ programming language.

The exact rules for the visibility of the variables are scripting language specific. Chapter 7 defines the rules for when the `language` attribute of the `page` directive is “java”.

2.8.2 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 2.8.3, “Implicit Objects); other objects are created explicitly through actions; objects can also be created directly using scripting code. The 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 the scripting elements through some scripting-level variables (see Section 2.8.3, “Implicit Objects).

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

Objects are always created within some JSP page instance that is responding to some *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. All 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 2.10.1, “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. All references to the object shall be released when the runtime environment reclaims the `ServletContext`. 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.

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

2.8.3 Implicit Objects

JSP page authors have access to certain implicit objects that are always available for use within scriptlets and expressions, without being declared first. 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 2-1.

TABLE 2-1 Implicit Objects Available in JSP Pages

Implicit Variable	Of Type	What It Represents	Scope
`request`	protocol dependent subtype of: `javax.servlet.ServletRequest` e.g: `javax.servlet.HttpServletRequest`	The request triggering the service invocation.	request
`response`	protocol dependent subtype of: `javax.servlet.ServletResponse` e.g: `javax.servlet.HttpServletResponse`	The response to the request.	page
`pageContext`	`javax.servlet.jsp.PageContext`	The page context for this JSP page.	page

TABLE 2-1 Implicit Objects Available in JSP Pages

Implicit Variable	Of Type	What It Represents	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
`application`	`javax.servlet.ServletContext`	The servlet context obtained from the servlet configuration object (as in the call `getServletConfig().getContext()`)	application
`out`	`javax.servlet.jsp.JspWriter`	An object that writes into the output stream.	page
`config`	`javax.servlet.ServletConfig`	The `ServletConfig` for this JSP page	page
`page`	`java.lang.Object`	the instance of this page's implementation class processing the current request[a]	page

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

In addition, in an error page, you can access the `exception` implicit object, described in TABLE 2-2.

TABLE 2-2 Implicit Objects Available in Error Pages

Implicit Variable	Of Type	What It Represents	scope
`exception`	`java.lang.Throwable`	The uncaught `Throwable` that resulted in the error page being invoked.	page

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 9.6.1 for some non-normative conventions for the introduction of new implicit objects.

2.8.4 The pageContext Object

A `PageContext` provides an object that 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 `JspWriter`s.

See Chapter 8 for more details.

2.9 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 2.6, “Quoting and Escape Conventions.

2.10 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 5 describes the mapping of directives into XML elements.

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 next chapter.

2.10.1 The `page` Directive

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

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). However, there shall be only one occurrence of any attribute/value defined by this directive in a given translation unit with the exception of the "`import`" attribute; multiple uses of this attribute are cumulative (with ordered set union semantics). Other such multiple attribute/value (re)definitions result in a fatal translation error.

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, indicates that the page is thread safe, and provides an error page.

```
<%@ page buffer="none" isThreadSafe="yes" 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 16K should be used.

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

2.10.1.1 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"          }
```

The details of the attributes are as follows:

`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 1.2, 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 7.

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 2.8.3, "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.

`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 6).

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 9.6.1 for an alternate way to introduce objects into a JSP page that does not have this drawback.

`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, i.e. 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 one in that package. The import list shall be imported by the translated JSP page implementation and are thus available to the scripting environment. The default import list is `java.lang.*`, `javax.servlet.*`, `javax.servlet.jsp.*` and `javax.servlet.http.*`. This value 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, and the suffix "kb" is mandatory. If a buffer size is specified (e.g `12kb`) 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`.

`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. The default is "`true`". Note: it is illegal to set `autoFlush` to "`false`" when "`buffer=none`".
`isThreadSafe`	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 page shared state. Default is "`true`". Note that even if the *isThreadSafe* attribute is "`false`" the JSP page author must ensure that access to any shared objects shared in either the `ServletContext` or the `HttpSession` are properly synchronized.
`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`"

`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 to for error processing.

The provided URL spec is as in Section 2.2.1.

The resource named has to be a JSP page in this version of the specification.

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`".

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.

When an error page is also indicated in the web.xml descriptor, the JSP error page applies first, then the web.xml page.

`contentType`

Defines the character encoding for the JSP page and for the response of the JSP page and the MIME type for the response of the JSP page.

Values are either of the form "TYPE" or "TYPE; charset=CHARSET" with an optional white space after the ";". CHARSET, if present, must be the IANA value for a character encoding. TYPE is a MIME type, see the IANA registry for useful values.

The default value for TYPE is "text/html"; the default value for the character encoding is ISO-8859-1.

See Section 3.1.2 for complete details on character encodings.

2.10.2 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 using the prefix introduced by the `taglib` directive.

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 9 for more specification details. And see Section B.2 for an implementation note.

Examples

In the following example, a tag library is introduced and made available to this page using the `super` prefix; no other tags 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>
```

2.10.2.1 Syntax

```
<%@ taglib uri="tagLibraryURI" prefix="tagPrefix" %>
```

where the attributes are:

`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 9.
`tagPrefix`	Defines the *prefix* string in `<prefix>:<tagname>` that is used to distinguish a custom action, e.g <myPrefix:myTag> prefixes jsp:, jspx:, java:, javax:, servlet:, sun:, and sunw: are reserved. Empty prefixes are illegal in this version of the specification.

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

2.11 Scripting Elements

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

There are three 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 2.10.1, “The page Directive). In JSP 1.2, 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. Scriplets 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 then 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 7.

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 5.4.

2.11.1 Declarations

Declarations are used to declare variables and methods in the scripting language used in a JSP page. A declaration should 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.

Examples

For example, the first declaration below declares an integer, and initializes it to zero; while the second declaration declares a method.

```
<%! int i = 0; %>
<%! public String f(int i) { if (i<3) return("..."); ... } %>
```

Syntax

<%! *declaration(s)* %>

2.11.2 Scriptlets

Scriptlets can contain any code fragments that are valid for the scripting language specified in the `language` directive. Whether the code fragment is legal depends on the details of the scripting language; see Chapter 7.

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

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

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

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
<% } %>
```

Syntax

<% *scriptlet* %>

2.11.3 Expressions

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

If the result of the expression cannot be coerced to a `String` then either a translation time error shall occur, or, 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. If so, they take effect when the expression is evaluated. Expressions are evaluated left-to-right in the JSP page. If the expressions appear 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 contents of an expression must be a complete expression in the scripting language in which they are written.

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

Examples

In the next example, the current date is inserted.

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

Syntax

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

2.12 Actions

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

The JSP specification includes some action types that are *standard* and must be implemented by all conforming JSP containers. New action types are introduced using the `taglib` directive.

The syntax for action elements is based on XML; the only transformation needed is due to quoting conventions (see Section 5.5).

2.13 Tag Attribute Interpretation Semantics

Generally, all custom and standard action attributes and their values either remain uninterpreted by, or have well defined action-type specific semantics known to, a conforming JSP container. However there are two exceptions to this general rule: some attribute values represent request-time attribute values and are processed by a conforming JSP container, and the id and scope attributes have special interpretation.

2.13.1 Request Time Attribute Values

Action elements (both standard and custom) can define named attributes and associated values. Typically a JSP page treats these values as fixed and immutable but the JSP 1.2 provides a mechanism to describe a value that is computed at request time.

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. Request-time attribute values can only be used in actions. If there are more than one such attribute in a tag, the expressions are evaluated left-to-right.

Only attribute values can be denoted this way (e.g. the name of the attribute is always an explicit name), and the expression must appear by itself (e.g. multiple expressions, and mixing of expressions and string constants are not permitted; instead perform these operations within the expression).

The resulting value of the expression depends upon the expected type of the attribute's value. The type of an action element indicates the valid Java programming languag type for each attribute value; the default is `java.lang.String`.

By default, all attributes have page translation-time semantics. Attempting to specify a scriptlet expression as a value for an attribute that 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 actions defined in the JSP 1.2 specification have page translation-time semantics.

The following attributes accept request-time attribute expressions:

- The value attribute of jsp:setProperty (2.13.2).
- The beanName attribute of jsp:useBean (2.13.1).
- The page attribute of jsp:include (2.13.4).

- The page attribute of jsp:forward (2.13.5).
- The value attribute of jsp:param (2.13.6).

Note – There is an anomaly: we cannot use a custom tag to provide the value of an argument to another custom tag. For example, one cannot use jsp:getProperty... There are several ways to address this.

2.13.2 The `id` Attribute

The `id="name"` attribute/value tuple in an element has special meaning to a JSP container, both 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.

- In addition, if the action type creates one or more object instance at client request processing time, one of these objects will usually be associated by the JSP container with the named value and can be accessed via that name in various contexts through the *`pagecontext`* object described later in this specification.

  Furthermore, the *`name`* is also used to expose a variable (name) in the page's scripting language environment. The scope of this scripting language dependent variable is dependent upon the scoping rules and capabilities of the actual scripting language used in the page. Note that this implies that the *`name`* value syntax shall comply with the variable naming syntax rules of the scripting language used in the page.

  Chapter 7 provides details for the case where the language attribute is `"java"`.

For example, the `<jsp:usebean id="name" class="className" .../>` action defined later herein uses this mechanism in order to, possibly instantiate, and subsequently expose the named JavaBeans component to a page at client request processing time.

For example:

```
<% { // 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)
%>
```

2.13.3 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, as follows:

`page`

The named object is available from the `javax.servlet.jsp.PageContext` for the current page.

This reference shall be discarded upon completion of the current request by the page body.

It is illegal to change the instance object associated, such that its runtime type is a subset of the type of the current object previously associated.

`request`

The named object is available from the current page's `ServletRequest` object using the `getAttribute(`*`name`*`)` method.

This reference shall be discarded upon completion of the current client request.

It is illegal to change the value of an instance object so associated, such that its runtime type is a subset of the type(s) of the object previously so associated.

`session`	The named object is available from the current page's `HttpSession` object (which can in turn be obtained from the `ServletRequest` object) using the `getValue(`*`name`*`)` method. This reference shall be discarded upon invalidation of the current session. It is Illegal to change the value of an instance object so associated, such that its new runtime type is a subset of the type(s) of the object previously so associated. Note it is a fatal translation error to attempt to use `session` scope when the JSP page so attempting has declared, via the `<%@ page ... %>` directive (see later) that it does not participate in a `session`.
`application`	The named object is available from the current page's `ServletContext` object using the `getAttribute(`*`name`*`)` method. This reference shall be discarded upon reclamation of the `ServletContext`. It is Illegal to change the value of an instance object so associated, such that its new runtime type is a subset of the type(s) of the object previously so associated.

CHAPTER 3

Localization Issues

This chapter will describe localization issues with JSP.

3.1 I18N Issues

I18N will be reviewed to track the Servlet 2.3 resolutions.

We expect the global file to be useful in configuring portions of the JSP page sources according to locales.

3.1.1 Specifying Content Types

A JSP page can use the `contentType` attribute of the page directive to indicate the content type of the response it provides to requests. Since this value is part of a directive, a given page will always provide the same content type. If a page determines that the response should be of a different content type, it should do so “early”, determine what other JSP page or Servlet will handle this request and it should forward the request to the other JSP page or Servlet.

A registry of content types names is kept by IANA. See:

ftp://venera.isi.edu/in-notes/iana/assignments/media-types/media-types

3.1.2 Delivering Localized Content

The Java Platform support for localized content is based on a uniform representation of text internally as Unicode 2.0 (ISO010646) characters and the support for a number of character encodings to and from Unicode.

Any Java Virtual Machine (JVM) must support Unicode and Latin-1 encodings but most support many more. The character encodings supported by the JVM from Sun are described at:

http://java.sun.com/products/jdk/1.1/docs/guide/intl/encoding.doc.html

The JSP 1.1 specification assumes that JSP pages that will deliver content in a given character encoding will be written in that character encoding. In particular, the `contentType` attribute of the `page` directive describes both the character encoding of the JSP page and the character encoding of the resulting stream.

The valid names to describe the character encodings are those of IANA. They are described at:

ftp://venera.isi.edu/in-notes/iana/assignments/character-sets

The `contentType` attribute must only be used when the character encoding is organized such that ASCII characters stand for themselves, at least until the `contentType` attribute is found. The directive containing the `contentType` attribute should appear as early as possible in the JSP page.

The default character set encoding is ISO-8859-1 (also known as latin-1).

A JSP container may use some implementation-dependent heuristics and/or structure to determine what is the expected character encoding of a JSP page and then verify that `contentType` attribute is as expected.

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

See Section B.1 for some implementation notes.

CHAPTER 4

Standard Actions and Directives

This chapter describes the standard actions of JSP 1.2. The chapter also describes the include directive, as it could[1] be described using the tag extension mechanism in JSP 1.2, while the page and taglib directives are more basic and cannot be described this way.

4.1 Standard Directives

The include directive is described here.

4.1.1 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 `.jsp` file. The included file is subject to the access control available to the JSP container. The `file` attribute is as in Section 2.2.1.

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 1.2 specification does not have a way of directing the JSP container that included files have changed.

1. Strictly speaking, the syntax cannot be described using the tag mechanism, but otherwise it can.

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" %>
```

4.1.1.1 Syntax

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

4.1.2 Including Data in JSP Pages

Including data is a significant part of the tasks in a JSP page. Accordingly, the JSP 1.2 specification has two include mechanisms suited to different tasks. A summary of their semantics is shown in TABLE 4-1.

TABLE 4-1 Summary of Include Mechanisms in JSP 1.2

Syntax	What	Phase	Spec	Object	Description	Section
<%@ include file=... %>	directive	translation-time	virtual	static	Content is parsed by JSP container.	4.1.1
<jsp:include page= />	action	request-time	virtual	static and dynamic	Content is not parsed; it is included in place.	4.2.4

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.

An include directive regards a resource like a JSP page as a static object; i.e. the bytes in the JSP page are 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.

4.2 Standard Actions

The JSP 1.2 specification defines some standard action types that are always available, regardless of the version of the JSP container or Web server the developer uses. The standard action types are in addition to any custom types specific to a given JSP container implementation.

4.2.1 <jsp:useBean>

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

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 it is not found it will attempt to create the object using the other attributes. It is also possible to use this action only to give a local name to an object define elsewhere, as in another JSP page or in a Servlet; this can be done by using the `type` attribute and not providing neither `class` nor `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 (in the Java platform sense) to `type;` failure to do so is a translation-time error.

The attribute `beanName` is the name of a Bean, as specified in the JavaBeans specification for an argument to the instantiate() method in java.beans.Beans. I.e. it is 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 instantiate() will be raised. The value of this attribute can be a request-time attribute expression.

The actions performed are:

1. Attempt to locate an object based on the attribute values (`id, scope`). The inspection is done appropriately synchronized per scope namespace to avoid non-deterministic behavior.

2. Define a scripting language variable with the given `id` in the current lexical scope of the scripting language of the specified `type` (if given) or `class` (if type is not given).

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 `useBean` action.

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

4. 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 `useBean` action.

5. 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 that class is instantiated, and 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 7 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 `useBean` action.

6. 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 7 is performed.

7. If the `jsp:useBean` element 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; if there is template text it 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. 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 this element; either because it was already created or because it is newly created.

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

In this 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 our 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”/>
```

4.2.1.1 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 is not restricted.

The <jsp:useBean> tag has the following 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
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.
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, that 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.

4.2.2 <jsp:setProperty>

The jsp:setProperty action sets the value of properties in a Bean. The name attribute denotes an object that 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 they are simple or indexed, their type, and 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 setProperty.

String constants and request parameter values can be used to assign values to any a type that has a PropertyEditor class. When that is the case, the setAsText(String) method is used. A conversion failure arises if the method throws an IllegalArgumentException.

String constantsand request parameter values can also be used to assing to the types as listed in TABLE 4-2; the conversion applied is that shown in the table.

Request-time expressions can be assigned to properties of any type; no automatic conversions will be performed.

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

A conversion failure leads to an error; the error may be at translation or at request-time.

TABLE 4-2 Valid assignments in jsp:setProperty

Property Type	Conversion on String Value
boolean or Boolean	As indicated in java.lang.Boolean.valueOf(String)
byte or Byte	As indicated in java.lang.Byte.valueOf(String)
char or Character	As indicated in java.lang.Character.valueOf(String)
double or Double	As indicated in java.lang.Double.valueOf(String)
int or Integer	As indicated in java.lang.Integer.valueOf(String)
float or Float	As indicated in java.lang.Float.valueOf(String)
long or Long	As indicated in java.lang.Long.valueOf(String)

Examples

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

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

The following element sets a property from a value

```
<jsp:setProperty name=”results” property=”row” value=”<%= i+1 %>” />
```

4.2.2.1 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 2.4.2.

```
propertyValue ::= expr_scriptlet[1]
```

The <`jsp:setProperty`> element has the following attributes:

`name`	The name of a Bean instance defined by a `<jsp:useBean>` element or some other element. The Bean instance must contain the property you want to set. The defining element must appear before the `<jsp:setProperty>` element in the same file.
`property`	The name of the Bean property whose value you want to set If you set *`propertyName`* 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.

1. See syntax for expression scriptlet “<%= ... %>”

`param`	The name of the request parameter whose value you want to give to a Bean property. The name of the request parameter usually comes from a Web form If you omit `param`, 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` element 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.

4.2.3 <jsp:getProperty>

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

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 - i.e. the truth is always the value held by the pageContext object

Examples

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

4.2.3.1 Syntax

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

The attributes are:

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

4.2.4 <jsp:include>

A `<jsp:include .../>` element provides for the inclusion of static and dynamic resources in the same context as the current page. See TABLE 4-1 for a summary of include facilities.

The resource is specified using a relativeURLspec that is interpreted in the context of the Web server (i.e. it is mapped).

An included page only has access to the `JspWriter` object and it cannot set headers. This precludes invoking methods like `setCookie`(). A request-time Exception will be raised if this constraint is not satisfied. 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.

If the page output is buffered then the buffer is flushed prior to the inclusion. See Section B.4 for an implementation note.

Examples

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

4.2.4.1 Syntax

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

and

```
<jsp:include page="urlSpec" flush="true">
       { <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:

`page`	The URL is a relative *urlSpec* is as in Section 2.2.1. 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. The default value is “false”.

4.2.5 <jsp:forward>

A `<jsp:forward page="urlSpec" />` element allows the runtime dispatch of the current request to a static resource, a JSP pages or a Java Servlet class in the same context as the current page. A jsp:forward effectively terminates the execution of the current page. The relative *urlSpec* is as in Section 2.2.1.

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 then the buffer is cleared prior to forwarding.

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 element might be used to forward to a static page based on some dynamic condition.

```
<% String whereTo = "/templates/"+someValue; %>
<jsp:forward page='<%= whereTo %>' />
```

4.2.5.1 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 effected by the specified attributes as follows:

`page`	The URL is a relative *urlSpec* is as in Section 2.2.1. Accepts a request-time attribute value (which must evaluate to a String that is a relative URL specification).

4.2.6 <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:plugin` elements.

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 2.2 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.

4.2.6.1 Syntax

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

This action has two mandatory attributes: name and value. Name indicates the name of the parameter, value, which may be a request-time expression, indicates its value.

4.2.7 <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:param>` elements indicate the 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 needs 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>
```

4.2.7.1 Syntax

```
<jsp:plugintype="bean|applet"
       code="objectCode"
       codebase="objectCodebase"
       { align="alignment"          }
       { archive="archiveList"      }
       { height="height"         }
       { hspace="hspace"         }
       { jreversion="jreversion" }
       { name="componentName"       }
       { vspace="vspace"            }
       { width="width"           }
       { nspluginurl="url"       }
       { iepluginurl="url"       } >
       { <jsp:params>
        { <jsp:param name="paramName" value=”paramValue" /> }+
        </jsp:params> }
       { <jsp:fallback> arbitrary_text </jsp:fallback> }
</jsp:plugin>
```

`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
`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
`vspace`	As defined by HTML spec
`title`	As defined by the HTML spec

`width`	As defined by HTML spec
`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.

CHAPTER 5

JSP Pages as XML Documents

This chapter defines a standard XML document for each JSP page.

All JSP pages have an equivalent XML document. This *equivalent* XML document is the view of the JSP page that is exposed to the translation phase (see below).

A JSP page can also be written directly as its equivalent XML document. Unlike in JSP 1.0 and JSP 1.1 containers, the XML document itself can be delivered to a JSP container for processing.

It is not valid to intermix “standard syntax” and XML syntax inside the same source file.

A JSP page (in either syntax) can include via a directive a JSP page in any syntax. I.e. within each unit one syntax is used but each unit can use either syntax.

5.1 Why an XML Representation

There are a number of reasons why it would be impractical to define JSP pages as XML documents when the JSP page is to be authored manually:

- An XML document must have a single top element; a JSP page is conveniently organized as a sequence of template text and elements.
- In an XML document all tags are “significant”; to “pass through” a tag, it needs to be escaped using a mechanism like CDATA. In a JSP page, tags that are undefined by the JSP specification are passed through automatically.

- Some very common programming tokens, like “<“ are significant to XML; the JSP specification provides a mechanism (the <% syntax) to “pass through” these tokens.

On the other hand, the JSP specification is not gratuitously inconsistent with XML: all features have been made XML-compliant as much as possible.

The hand-authoring friendliness of JSP pages is very important for the initial adoption of the JSP technology; this is also likely to remain important in later time-frames, but tool manipulation of JSP pages will take a stronger role then. In that context, there is an ever growing collection of tools and APIs that support manipulation of XML documents.

The JSP 1.2 specification addresses both requirements by providing a friendly syntax and also defining a standard XML document for a JSP page.

5.2 Document Type

5.2.1 The jsp:root Element

An XML document representing a JSP page has `jsp:root` as its root element type. The root is also the place where taglibs will insert their namespace attributes. The top element has an `xmlns` attribute that enables the use of the standard elements defined in the JSP 1.1 specification.

```
<jsp:root
     xmlns:jsp=”http://java.sun.com/products/jsp/dtd/jsp_1_2.dtd”>
       remainder of transformed JSP page
</jsp:root>
```

5.2.2 Public ID

The proposed Document Type Declaration is:

```
<! DOCTYPE root
    PUBLIC“-//Sun Microsystems Inc.//DTD JavaServer Pages Version 1.1//EN”
       “http://java.sun.com/products/jsp/dtd/jspcore_1_2.dtd”>
```

5.3 Directives

A directive in a JSP page is of the form

<%@ *directive { attr=”value” }** %>

Most directives get translated into an element of the form:

<jsp:directive.*directive { attr=”value” }** />

5.3.1 The `page` directive

In the XML document corresponding to JSP pages, the `page` directive is represented using the syntax:

<jsp:directive.page *page_directive_attr_list* />

See Section 2.10.1 for description of *`page_directive_attr_list`*.

Example

The directive:

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

corresponds to the XML element:

```
<jsp:directive.page info=”my latest JSP Example” />
```

5.3.2 The `include` Directive

In the XML document corresponding to JSP pages, the include directive is represented using the syntax:

<jsp:directive.include file="*relativeURLspec”* flush="true|false" />

Examples

Below are two examples, one in JSP syntax, the other using XML syntax:

```
<%@ include file="copyright.hmtl" %>
<jsp:directive.include file="htmldocs/logo.html" />
```

5.3.3 The `taglib` Directive

In the XML document corresponding to JSP pages, the taglib directive is represented as an xmlns: attribute within the root element of the JSP page document.

5.4 Scripting Elements

The JSP 1.2 specification has three scripting language elements—declarations, scriptlets, and expressions. The scripting elements have a "<%"-based syntax as follows:

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

5.4.1 Declarations

In the XML document corresponding to JSP pages, declarations are represented using the syntax:

```
<jsp:declaration> declaration goes here </jsp:declaration>
```

For example, the second example from Section 2.11.1:

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

is translated using a CDATA statement to avoid having to quote the "<" inside the jsp:declaration.

```
<jsp:declaration> <![CDATA[ public String f(int i) { if (i<3)
return("..."); } ]]> </jsp:declaration>
```

DTD Fragment

```
<!ELEMENT jsp:declaration (#PCDATA) >
```

5.4.2 Scriptlets

In the XML document corresponding to JSP pages, directives are represented using the syntax:

```
<jsp:scriptlet> code fragment goes here </jsp:scriptlet>
```

DTD Fragment

```
<!ELEMENT jsp:scriptlet (#PCDATA) >
```

5.4.3 Expressions

In the XML document corresponding to JSP pages, directives are represented using the syntax:

```
<jsp:expression> expression goes here </jsp:expression>
```

DTD Fragment

```
<!ELEMENT jsp:expression (#PCDATA) >
```

5.5 Actions

The syntax for action elements is based on XML; the only transformations needed are due to quoting conventions and the syntax of request-time attribute expressions.

5.6 Transforming a JSP Page into an XML Document

The standard XML document for a JSP page is defined by transformation of the JSP page.

- Add a `<jsp:root>` element as the root. Enable a “jsp” namespace prefix for the standard tags within this root.
- Convert all the <% elements into valid XML elements as described in Section 5.4.1 and following sections.
- Convert the quotation mechanisms appropriately.
- Convert the `taglib` directive into namespace attributes of the `<jsp:root>` element.
- Create CDATA elements for all segments of the JSP page that do not correspond to JSP elements.

A quick summary of the transformation is shown in TABLE 5-1:

TABLE 5-1 XML standard tags for directives and scripting elements

JSP page element	XML equivalent
<%@ page ... %>	<jsp:directive.page ... />
<%@ taglib ... %>	jsp:root element is annotated with namespace information.
<%@ include ... %>	<jsp:directive.include .../>
<%! ... %>	<jsp:declaration> .... </jsp:declaration>
<% ... %>	<jsp:scriptlet> .... </jsp:scriptlet>
<%= .... %>	<jsp:expression> .... </jsp:expression>

5.6.1 Quoting Conventions

The quoting rules for the JSP 1.2 specification are designed to be friendly for hand authoring, they are not valid XML conventions.

Quoting conventions are converted in the generation of the XML document from the JSP page. This is not yet described in this version of the specification.

5.6.2 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.

Note – The JSP 1.1 syntax does not allow the use of custom actions to construct attribute values. This is issue # 4.

5.7 DTD for the XML document

The following is a DTD for the current XML mapping:

FIGURE 5-1 DTD for the XML document

```
<!ENTITY % jsp.body “
(#PCDATA
|jsp:directive.page
|jsp:directive.include
|jsp:scriptlet
|jsp:declaration
|jsp:expression
|jsp:include
|jsp:forward
|jsp:useBean
|jsp:setProperty
|jsp:getProperty
|jsp:plugin
|jsp:fallback
|jsp:params
|jsp:param)*
“>

<!ELEMENT jsp:useBean %jsp.body;>
<!ATTLIST jsp:useBean
id     ID   #REQUIRED
class  CDATA#REQUIRED
scope  (page|session|request|application) “page”>

<!ELEMENT jsp:setProperty EMPTY>
<!ATTLIST jsp:setProperty
name   IDREF#REQUIRED
propertyCDATA#REQUIRED
value  CDATA#IMPLIED
param  CDATA#IMPLIED>
```

```
<!ELEMENT jsp:getProperty EMPTY>
<!ATTLIST jsp:getProperty
name   IREF #REQUIRED
propertyCDATA#REQUIRED>

<!ELEMENTjsp:includeEMPTY>
<!ATTLISTjsp:include
flush  (true|false)"false"
page   CDATA#REQUIRED>

<!ELEMENT jsp:forward EMPTY>
<!ATTLISTjsp:forward
page   CDATA#REQUIRED>

<!ELEMENT jsp:scriptlet (#PCDATA)>

<!ELEMENT jsp:declaration (#PCDATA)>

<!ELEMENT jsp:expression (#PCDATA)>

<!ELEMENT jsp:directive.page EMPTY>
<!ATTLIST jsp:directive.page
languageCDATA“java”
extendsCDATA#IMPLIED
contentTypeCDATA“text/html; ISO-8859-1”
import CDATA#IMPLIED
session(true|false)“true”
buffer CDATA“8kb”
autoFlush(true|false)“true”
isThreadSafe(true|false)“true”
info   CDATA#IMPLIED
errorPageCDATA#IMPLIED
isErrorPage(true|false)“false”>

<!ELEMENT jsp:directive.include EMPTY>
<!ATTLIST jsp:directive.include
file CDATA #REQUIRED>

<!ELEMENT jsp:root %jsp.body;>
<!ATTLIST jsp:root
xmlns:jspCDATA#FIXED “http://java.sun.com/products/jsp/dtd/
jsp_1_0.dtd”>
```

CHAPTER 6

The JSP Container

This chapter provides details on the contracts between a JSP container and a JSP page.

This chapter is independent on the Scripting Language used in the JSP page. Chapter 7 provides the details specific to when the `language` directive has “java” as its value.

This chapter also presents the precompilation protocol (see Section 6.4).

JSP page implementation classes should use the JspFactory and PageContext classes so they will take advantage of platform-specific implementations.

6.1 The 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 implements 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 in the process some other objects. A JSP page may also indicate how some events (in JSP 1.1 only *init* and *destroy* events) are to be handled.

The Protocol Seen by the Web Server

It is the role of the JSP container to first locate the appropriate instance of the JSP page implementation class and then to deliver requests to it according to the Servlet protocol. As indicated elsewhere, a JSP container may need to create such a class dynamically from the JSP page source before delivering a request and response objects to it.

Thus, Servlet 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 pages use the HTTP protocol, but other protocols are allowed by this specification.

The JSP container automatically makes available to the JSP page implementation object a number of server-side objects. See Section 2.8.3.

The Protocol Seen by the JSP Page Author

The JSP specification also defines the contract between the JSP container and the JSP page author. This is, what assumptions can an author 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 is provided in Chapter 7.

The contract also describes how a JSP author can indicate that some actions must be taken when the *init()* and *destroy()* methods of the page implementation occur. In JSP 1.1 this is done by defining methods with name *jspInit()* and *jspDestroy()* in a declaration scripting element in the JSP page. Before the first time a request is delivered to a JSP page a *jspInit()* method, if present, will be called to prepare the page. Similarly, a JSP container can reclaim the resources used by a JSP page at any time that a request is not being serviced by the JSP page by invoking first its *jspDestroy()* method, if present.

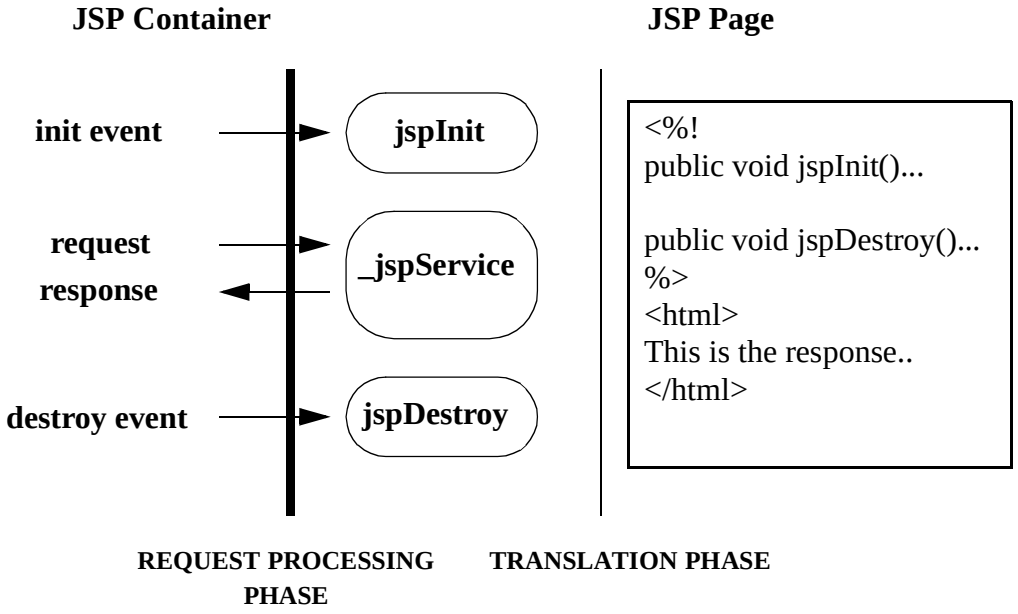
A JSP page author may **not** (re)define any of the Servlet methods through a declaration scripting element.

The JSP specification reserves the semantics of methods and variables starting with jsp, _jsp, jspx and _jspx, in any combination of upper and lower case.

The HttpJspPage Interface

The enforcement of the contract between the JSP container and the JSP page author is aided by requiring that the Servlet class corresponding to the JSP page must implement the `HttpJspPage` interface (or the `JspPage` interface if the protocol is not HTTP).

FIGURE 6-1 Contracts between a JSP Page and a JSP Container.



The involved contracts are shown in FIGURE 6-1. We now revisit this whole process in more detail.

6.2 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 is implementation-dependent, and may even vary between one JSP and another, so minimal assumptions should be made. One implication of this is that classes in the unnamed package should not be used without an explicit “import” of the class

The creation of the implementation class for a JSP page may be done solely by the JSP container, or it may involve a superclass provided by the JSP page author through the use of the *extends* attribute in the *jsp* directive.

The extends mechanism is available for sophisticated users and it should be used with extreme care as it restricts what some of the decisions that a JSP container can take, e.g. to improve performance.

The JSP page implementation class will implement `Servlet` and the `Servlet` protocol will be used to deliver requests to the class.

A JSP page implementation class may depend on some support classes; if it does, and the JSP page implementation class is packaged into a WAR, those classes will have to be included in the packaged WAR 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 then guarantee that requests from and responses to 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`.

6.2.1 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 specification for details.

The contract between the JSP container and the JSP page author is described in TABLE 6-1. The responsibility for adhering to this contract rests only on the JSP container implementation if the JSP page does not use the *extends* attribute of the *jsp* directive; otherwise, the JSP page author guarantees that the superclass given in the extends attribute supports this contract.

TABLE 6-1 How the JSP Container Processes JSP Pages

Comments	Methods the JSP Container Invokes

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	`void jspInit()`
Method is optionally defined in JSP page. Method is invoked before destroying the page.	`void jspDestroy()`
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.	`void _jspService(<ServletRequestSubtype>, <ServletResponseSubtype>) throws IOException, ServletException`

6.2.2 Request and Response Parameters

As shown in TABLE 6-1, 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 contract for HTTP 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 implement the `request` and `response` interfaces for the HTTP protocol as described in this section.

6.2.3 Omitting the `extends` Attribute

If the `extends` attribute of the `language` directive (see Section 2.10.1, “The page Directive) in a JSP page is not used, the JSP container can generate any class that satisfies the contract described in TABLE 6-1 when it transforms the JSP page.

In the following code examples, CODE EXAMPLE 6-1 illustrates a generic HTTP superclass named `ExampleHttpSuper`. CODE EXAMPLE 6-2 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 needs not discover if the JSP page includes a declaration with `jspInit()` or `jspDestroy();` this simplifies very significantly the implementation.

CODE EXAMPLE 6-1 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();
}
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, resonse);


/**
* 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 6-2 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

        HttpSession session = request.getSession();
        ServletContext context =
            getServletConfig().getServletContext();


        // for this example, we assume a buffered directive

        JSPBufferedWriter out = new
            JSPBufferedWriter(response.getWriter());

        // next is code from scriptlets, expressions, and static text.

    }

}
```

6.2.4 Using the `extends` Attribute

If the JSP page author uses `extends`, the generated class is identical to the one shown in CODE EXAMPLE 6-2, 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 as `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.

6.3 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, none of the operations that rely on headers, such as the `setContentType`, `redirect`, or `error` methods are valid until the `flush` method is executed and the headers are sent.

The `javax.servlet.jsp.JspWriter` class buffers and sends output. The `JspWriter` class is used in the `_jspPageService` 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();
}
```

You can find the complete listing of `javax.servlet.jsp.JspWriter` in Chapter 8.

With buffering turned on, you can still use a redirect method in a scriptlet in a `.jsp` file, by invoking `response.redirect(someURL)` directly.

6.4 Precompilation

A JSP page that is using the HTTP protocol will receive HTTP requests. JSP 1.2 compliant containers must support a simple *precompilation protocol*, as well as some basic *reserved parameter names*. Note that the precompilation protocol should not be confused with the notion of compiling a JSP page into a Servlet class (Appendix A).

6.4.1 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_"

6.4.2 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 hint to the JSP container to *precompile* the JSP page into its JSP page implementation class. The hint is conveyed by given the parameter the value "true" or no value, but note that the request can be just ignored in all cases.

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 be delivered to the page with no changes.

6. ?jsp_precompile="foo"

This is illegal and will generate an HTTP error; 500 (Server error).

CHAPTER 7

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 details of the relationship between the scripting declarations, scriptlets, and scripting expressions and the Java programming language is explained in detail in the following sections. The description is in terms of the structure of the JSP page implementation class; recall that a JSP container need not necessarily generate the JSP page implementation class but it must behave as if one existed.

7.1 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 just language fragments, not complete language statements.

Valid JSP Page

A JSP page is valid for a Java Platform if and only if the JSP page implementation class defined by TABLE 7-1 (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.

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.

Implementation Flexibility

The transformations described in this Chapter need not be performed literally; an implementation may want to implement things differently to provide better performance, lower memory footprint, or other implementation attributes.

TABLE 7-1 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 jsp directive. Name of class (_jspXXX) is implementation dependent.	`class _jspXXX extends SuperClass`
Start of body of JSP page implementation class	`{`
(1) Declaration Section	`// declarations ...`

signature for generated method	`public void _jspService(<ServletRequestSubtype> request,`
	`  <ServletResponseSubtype> response)`
	`  throws ServletException, IOException {`

(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	`}`

7.2 Declarations Section

The declarations section correspond 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.

7.3 Initialization Section

This section defines and initializes the implicit objects available to the JSP page. See Section 2.8.3, “Implicit Objects.

7.4 Main Section

This section provides the main mapping between a request and a response object.

The contents of code segment 2 is determined from scriptlets, expressions, and the text body of the JSP page. These elements are processed sequentially; a translation for each one is determined as indicated below, and its translation is inserted into this section. The translation depends on the element type:

1. *Template data* is transformed into code that will place the template data into the stream currently named by the implicit variable `out`. All white space is preserved.

   Ignoring quotation issues and performance issues, this corresponds to a statement of the form:

   out.print(*template*);

2. A *scriptlet* is transformed into its Java statement fragment.

3. 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 currently 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:

   out.print(*expression*);

4. An action defining one or more objects is transformed into one or more variable declarations for these objects, together with code that initializes these variables. The visibility of these variables is affected by other constructs, like the scriptlets.

   The semantics of the action type determines the name of the variables (usually that of the `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 that of the `id` attribute, its type is that of the `class` attribute.

   Note that the value of the `scope` attribute does not affect the visibility of the variables within the generated program, it only affects where (and thus for how long) there will be additional references to the object denoted by the variable.

CHAPTER 8

Core API

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.

8.1 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.

8.1.1 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.

JspPage objects are obtained from the JspFactory object.

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 apropriate 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 authomatically 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;
```

8.1.1.1 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 getServletConfig() 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

8.1.2 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.

HttpJspPage objects are obtained from the JspFactory class.

The behaviour is identical to that of the JspPage, except for the signature of the _jspService method, which is now expressable in the Java type system and included explicitly in the interface.

See Also: JspPage

8.1.2.1 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.

Throws:
`IOException`, `ServletException`

8.1.3 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.

8.1.3.1 Constructors

```
public JspFactory()
```

8.1.3.2 Methods

```
public static synchronized JspFactory getDefaultFactory()
```

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() mehtod 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

`config` - the ServletConfig for 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:
`default` - The default factory implementation

8.1.4 JspEngineInfo

Syntax

```
public abstract class JspEngineInfo
```

Description

The JspEngineInfo is an abstract class that provides information on the current JSP engine.

8.1.4.1 Constructors

```
public JspEngineInfo()
```

8.1.4.2 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

8.2 Implicit Objects

The `PageContext` object and the `JspWriter` are available by default as implicit objects.

8.2.1 PageContext

Syntax

```
public abstract class PageContext
```

Description

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.

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 implementation 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()` and `popBody()`

Methods Intended for JSP authors

Some methods provide **uniform access** to the diverse scopes objects. The implementation must use the underlying Servlet machinery corresponding to that scope, so information can be passed back and forth between Servlets and JSP pages. The methods are: `setAttribute()`, `getAttribute()`, `findAttribute()`, `removeAttribute()`, `getAttributesScope()` and `getAttributeNamesInScope()`.

The following methods provide **convenient access** to implicit objects: `getOut()`, `getException()`, `getPage()` `getRequest()`, `getResponse()`, `getSession()`, `getServletConfig()` and `getServletContext()`.

The following methods provide support for **forwarding, inclusion and error handling**: `forward()`, `include()`, and `handlePageException()`.

8.2.1.1 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 tables

`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 ServletRequest associated with the Servlet that 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.

8.2.1.2 Constructors

`public` **`PageContext`**`()`

8.2.1.3 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.

Returns: the value associated or null

`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:
`ServletException`, `IOException`

`IllegalArgumentException` - if target resource URL is unresolvable

`IllegalStateException` - if `ServletResponse` is not in a state where a forward can be performed

`SecurityException` - if target resource cannot be accessed by caller

`public abstract java.lang.Object` **`getAttribute`**`(java.lang.String name)`

Return the object associated with the name in the page scope or null if not found.

Parameters:
`name` - the name of the attribute to get

Throws:
`NullPointerException` - if the name is null

`IllegalArgumentException` - if the scope is invalid

`public abstract java.lang.Object` **`getAttribute`**`(java.lang.String name, int scope)`

Return the object associated with the name in the specifed 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

Throws:
`NullPointerException` - if the name is null

`IllegalArgumentException` - if the scope is invalid

`public abstract java.util.Enumeration` **`getAttributeNamesInScope`**`(int scope)`

Enumerate all the attributes in a given scope

Returns: an enumeration of names (java.lang.String) of all the attributes the specified scope

`public abstract int` **`getAttributesScope`**`(java.lang.String name)`

Get the scope where a given attribute is defined.

Returns: the scope of the object associated with the name specified or 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 JspWriter` **`getOut`**`()`

The current value of the out object (a JspWriter).

Returns: the current JspWriter stream being used for client response

`public abstract java.lang.Object` **`getPage`**`()`

The current value of the page object (a Servlet).

Returns: the Page implementation class instance (Servlet) 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 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.

TODO - should handle Throwable

Parameters:
`e` - the exception to be handled

Throws:
`ServletException`, `IOException`

`NullPointerException` - if the exception is null

`SecurityException` - if target resource cannot be accessed by caller

```
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`, `IOException`

`IllegalArgumentException` - if the target resource URL is unresolvable

`SecurityException` - if target resource cannot be accessed by caller

```
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 emthod 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 wihtin 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 libray 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`

`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 PageConxtext

Returns: the saved JspWriter.

`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 libray authors.

`public abstract void` **`removeAttribute`**`(java.lang.String name)`

Remove the object reference associated with the given name, look in all scopes in the scope order.

Parameters:
`name` - The name of the object to remove.

`public abstract void` **`removeAttribute`**`(java.lang.String name, int scope)`

Remove the object reference associated with the specified name in the given scope.

Parameters:
`name` - The name of the object to remove.

`scope` - The scope where to look.

`public abstract void` **`setAttribute`**`(java.lang.String name, java.lang.Object attribute)`

Register the name and object specified with page scope semantics.

Parameters:
`name` - the name of the attribute to set

`attribute` - the object to associate with the name

Throws:
`NullPointerException` - if the name or object is null

`public abstract void` **`setAttribute`**`(java.lang.String name, java.lang.Object o, int scope)`

register the name and object specified with appropriate scope semantics

Parameters:
`name` - the name of the attribute to set

`o` - the object to associate with the name

`scope` - the scope with which to associate the name/object

Throws:
`NullPointerException` - if the name or object is null

`IllegalArgumentException` - if the scope is invalid

8.2.2 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 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 when 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; a case may be 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 has the disadvantage that runaway computations may 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 fluch 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 opertion would result in a buffer overflow condition.

See Also: `java.io.Writer`, `java.io.BufferedWriter`, `java.io.PrintWriter`

8.2.2.1 Fields

`protected boolean` **`autoFlush`**

`protected int` **`bufferSize`**

`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

8.2.2.2 Constructors

`protected` **`JspWriter`**`(int bufferSize, boolean autoFlush)`

protected constructor.

8.2.2.3 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 mehtod 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 translated into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the `java.io.Writer.write(int)` method.

Parameters:
`b` - The `boolean` to be printed

Throws:
`java.io.IOException`

`public abstract void` **`print`**`(char c)`

Print a character. The character is translated into one or more bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the `java.io.Writer.write(int)` method.

Parameters:
`c` - The `char` to be printed

Throws:
`java.io.IOException`

`public abstract void` **`print`**`(char[] s)`

Print an array of characters. The characters are converted into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the `java.io.Writer.write(int)` method.

Parameters:
`s` - The array of chars to be printed

Throws:
`NullPointerException` - If `s` is `null`

`java.io.IOException`

`public abstract void **print**(double d)`

Print a double-precision floating-point number. The string produced by `java.lang.String.valueOf(double)` is translated into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the `java.io.Writer.write(int)` method.

Parameters:
`d` - The `double` to be printed

Throws:
`java.io.IOException`

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 translated into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the `java.io.Writer.write(int)` method.

Parameters:
`f` - The `float` to be printed

Throws:
`java.io.IOException`

See Also: `java.lang.Float`

`public abstract void **print**(int i)`

Print an integer. The string produced by `java.lang.String.valueOf(int)` is translated into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the `java.io.Writer.write(int)` method.

Parameters:
`i` - The `int` to be printed

Throws:
`java.io.IOException`

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 translated into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the `java.io.Writer.write(int)` method.

Parameters:
`l` - The `long` to be printed

Throws:
`java.io.IOException`

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 translated into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the `java.io.Writer.write(int)` method.

Parameters:
`obj` - The `Object` to be printed

Throws:
`java.io.IOException`

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 converted into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the `java.io.Writer.write(int)` method.

Parameters:
`s` - The `String` to be printed

Throws:
`java.io.IOException`

`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`

`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()`.

Throws:
`java.io.IOException`

`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()`.

Throws:
`java.io.IOException`

`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()`.

Throws:
`java.io.IOException`

`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()`.

Throws:
`java.io.IOException`

`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()`.

Throws:
`java.io.IOException`

`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()`.

Throws:
`java.io.IOException`

`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()`.

Throws:
`java.io.IOException`

`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()`.

Throws:
`java.io.IOException`

`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()`.

Throws:
`java.io.IOException`

8.3 An Implemention Example

An instance of an implementation dependent subclass of this abstract base class can be created by a JSP implementation class at the begining 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);
    }
}
```

8.4 Exceptions

The `JspException` class is the base class for all JSP exceptions. The `JspTagException` is used by the tag extension mechanism.

8.4.1 JspException

Syntax

```
public class JspException extends java.lang.Exception
```

Direct Known Subclasses: JspTagException

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.

8.4.1.1 Constructors

`public` **`JspException`**`()`

Construct a JspException

`public` **`JspException`**`(java.lang.String msg)`

An exception with a message

8.4.2 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.

8.4.2.1 Constructors

`public` **`JspTagException`**`()`

No message

`public` **`JspTagException`**`(java.lang.String msg)`

Constructor with a message.

CHAPTER 9

Tag Extensions

This chapter describes the mechanisms for introducing new actions into a JSP page. The mechanisms include a portable run-time support, a validation mechanism, and authoring tool support, all bundled into the concept of a tag library.

The tag library concept was introduced in the JSP 1.1 specification and it incorporated run-time support, some validation, and minimal tool authoring support. The JSP 1.2 specification has added some refinements to the run-time and authoring tool support, significantly extended the validation step.

All JSP 1.1 tag libraries run, unchanged and without any change in behavior, in a JSP 1.2 container.

This chapter provides an overview of the mechanism and describes the Tag Library Descriptor, and the taglib directive. The detailed description of the APIs involved follows in Chapter 10.

9.1 Introduction

A Tag Library abstracts some functionality 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, and can then be used in the page using the prefix given by the directive. An action can create new objects that can then be passed to other actions or can be manipulated programmatically through an scripting element in the JSP page.

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 and modify a JSP page.
- *Validate* the JSP page.
- Present the JSP page to the end user.

A Tab Library is described via a Tag Library Descriptor (a TLD), an XML document that is described further below.

9.1.1 Goals

The tag extension mechanism described in this chapter addresses the following goals:

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. We would like to make it very easy for vendors of functionality to expose it through actions.

Expressive - We want to enable a wide range of actions to be described in this mechanism, including:

- Nested actions.
- Scripting elements inside the body.
- Creation, use and updating of scripting variables.

Usable from different scripting languages - Although the JSP specification currently only defines the semantics for scripting based on the Java programming language, we want to leave open other scripting languages.

Building upon existing concepts and machinery- We do not want to reinvent machinery that exists elsewhere. Also, we want to avoid future conflicts whenever we can predict them.

9.1.2 Overview

The processing of a JSP page conceptually follows this steps:

- **Parsing**

First parse the JSP syntax, processing include directives. This produces an XML document annotated with debug information related to the original JSP source. Information in the TLD is needed to process the JSP document, including identifying the custom tags, so this step requires some processing of the taglib directives.

The XML document can be input directly too. If the XML document is entered directly, the custom tags are indicated as such.

- **Validation**

  The tag libraries in the XML document are processed in the order they appear.

  Each library is checked for a validator class and, if present, the whole document is made available to the validation method as a PageInfo object.

  Then, each custom tag in the library is checked to see if there are TagExtraInfo classes, and, if so, the isValid() method will be consulted.

- **Translation**

  Finally the XML document is processed to create a JSP page implementation class. This process may involve creating scripting variables. Each custom action may provide this information, either statically in the TLD, or in a more flexible manner 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 and requests will be directed to an instance of the class. At run-time tag handler instances will be created and methods will be invoked in them.

Tag Handlers

The JSP page implementation class instantiates *tag handlers* which are the basic runtime mechanism for defining the semantics of custom actions.

A tag handler is a Java class that implements the `Tag` or `BodyTag` interfaces and that is the run-time representation of a custom action.

The JSP page implementation class instantiates (or reuses) a tag handler object for each action in the JSP page. This 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`.

- `Tag` defines the basic methods that are needed in all tag handlers. These methods include setter methods to initialize a tag handler with context data and with the attribute values of the corresponding action, and the two methods: `doStartTag()` and `doEndTag()`.

- `IterationTag` is an extension to Tag that provides one additional method: `doAfterBody`() for requiring the reevaluation of the body of the tag.
- `BodyTag` is an extension of IterationTag with two news methods for when the tag handler wants to manipulate its body: `setBodyContent`() passes a buffer (the `BodyContent` object), and `doInitBody()` provides an opportunity to do some activity on that buffer before the first evaluation of the body into the buffer.

The use of interfaces simplifies taking an existing Java object and making it a tag handler. There are also two support classes that can be used as base classes: `TagSupport` and `BodyTagSupport`.

Event Listeners

A tag library may include some classes that are event listeners (see the Servlet 2.3 specification). The listeners are listed in the tag library descriptor and the JSP container will automatically instantiate the listener classes and register them in a way analogous to how it is done in web.xml. Essentially, the mechanism just locates the TLDs in the Web Application (be them in WEB-INF/classes or in WEB-INF/lib), reads their <listener> elements and regards them as an extension of those listed in web.xml.

The order in which the listeners are registered is undefined.

9.1.3 Simple Examples

Next we describe a few prototypical uses of tag extensions, briefly sketching how they take advantage of these mechanisms.

Simple Actions

The simplest type of action just *does something*, perhaps with some parameters to modify what the “something” is, and improve reusability.

This type of action can be implemented with a tag handler that just implements the Tag interface. The tag handler only needs to use the tag handler’s method `doStartTag()`. The method is invoked when the start tag is encountered and can access the attributes of the tag and may also want to access information on the state of the JSP page; this information is passed to the Tag object before the call to `doStartTag()` through several setter method calls.

Since simple actions with an empty body are common, the Tag Library Descriptor can be used to indicate that the tag is always intended to be empty; this leads to better error checking at translation time and to better code quality in the JSP page implementation class.

Actions with a Body

Another set of quite simple actions require something to happen when the start tag is found, and then when the end tag is found. The Tag interface can also be used. The `doEndTag()` is similar to `doStartTag()`, except that it is invoked when the end tag of the action is encountered. The result of the `doEndTag` invocation indicates whether the remaining of the page is to be evaluated or not.

Conditionals

In some cases, a body needs to be invoked only when some conditions happen. This is still supported by the basic Tag interface, through the use of return values in the doStartTag() method.

Iterations

The `Tag` protocol cannot be used to do iteration. For iteration the additional `IterationTag` interface is needed. The `doAfterBody()` method is invoked to determine whether to reevaluate the body or not.

Actions that Process their Body

Consider an example of an action that will take its body, and reevaluate it many times, creating a stream of response data. The IterationTag protocol is used for this. But, if the result of the reinterpretation is to be further manipulated, for whatever reason, including just discarding it, we need some way to divert the potential output of the those reevaluations. This is done through the creation of a `BodyContent` object, and the use of the `setBodyContent()` method, which is part of the `BodyTag` interface. `BodyTag` also provides another method `doInitBody()` which is invoked just after setBodyContent() but before the first body evaluation to provide an opportunity to interact with the body.

Cooperating Actions

Often the best way to describe some functionality is through several cooperating actions. For example, an action may be used to describe information that leads to the creation of some server-side object, while another action may use that object elsewhere in the page. One way for these actions to cooperate is explicitly, via using scripting variables: one action creates an object and gives it a name, the other refers to it through this name. Scripting variables are discussed briefly below.

Two actions can also cooperate implicitly using different conventions. For example, perhaps the last action applies, or perhaps there is only one action of a given type per JSP page. A more flexible and very convenient mechanism for action cooperation is using the nesting structure to describe scoping. Each tag handler is told of its parent tag handler (if any) using a setter method; the `findAncestorWithClass` static method in `TagSupport` can then be used to locate a tag handler with some given properties.

Actions Defining Scripting Variables

A custom action may create some server-side objects and make them available to the scripting elements by creating or updating some scripting variables. The specific variables thus effected are part of the semantics of the custom action and are the responsability of the tag library author. This information is used at JSP page translation time and can be described in one of two ways, either directly in the TLD for simple cases, or through subclasses of `TagExtraInfo`. Either mechanism indicates what are the names and types of the scripting variables. At request time the tag handler will associate objects to these scripting variables through the pageContext object. It is the responsibility of the JSP page translator to automatically supply all the required code to do the “synchronization” between the pageObject values and the scripting variables.

9.2 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 9.3 below.

9.2.1 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 packaged so the JAR file must have a tag library descriptor file named META-INF/taglib.tld.

9.2.2 Location of Java Classes

A tag library contains classes that are intended to be instantiated at translation time and classes that are intended to be instantiated at request time. The first ones include TagLibraryValidator and TagExtraInfo classes. The second ones include tag handler and event listener classes. All these classes are treated as any other Java class: in a Web Application they must reside in the standard locations for Java classes: either in a JAR file in the WEB-INF/lib directory or in a directory in the WEB-INF/classes directory.

The previous rule indicates that a JAR containing a packaged tag libraries can be dropped into the WEB-INF/lib directory to make its classes available at request time (and also at translation time, see Section 9.3.4). The mapping between the URI and the TLD is explained further below.

9.2.3 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 that will distinguish 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 first resolves the URI into a TLD resource path, and then it derives the TLD object itself from the TLD resource path.

If a 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 the `taglib` directive.

9.3 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 include a TagLibraryValidator class that can validate that a JSP page conforms to whatever set of constraints are expected by the tag library.

Each action in the library is described by giving its name, the class for its tag handler, optional information on the scripting variables created by the action, and information on all the attributes of the action. Scripting variable information can be given directly in the TLD or through a `TagExtraInfo` class. Each valid attribute is mentioned explicitly, with indication on whether it is mandatory or not, whether it can accept request-time expressions, and additional information.

A TLD file is useful as a descriptive mechanism for providing information on a Tag Library. It has the advantage that it can be read by tools without having to instantiate objects or load classes. The approach we follow conforms to the conventions used in other J2EE technologies.

The DTD to the tag library descriptor is organized so that interesting elements have an optional ID attribute. This attribute can be used by other documents, like vendor-specific documents, to provide annotations of the TLD information. An alternative approach, based on XML name spaces have some interesting properties but it was not pursued in part for consistency with the rest of the J2EE descriptors.

The official DTD is described at "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_2.dtd"

9.3.1 TLD resource path

A URI in a taglib directive is mapped into a relative URI specification (as in section 2.5.2, i.e. a URL without a protocol and host components) that starts with “/” that is called the TLD resource path.

The TLD resource path is to be interpreted relative to the root of the Web Application and should resolve to either a TLD file directly, or to a JAR file with a TLD 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 two mechanisms: a map in `web.xml`, and a default mapping that is to be used if the map does not contain the URI. The default mapping is designed for more casual use of the mechanism.

9.3.2 Taglib map in web.xml

The `web.xml` file can include a map between URIs and TLD resource paths. The map is described using the `taglib` element of the Web Application Deployment descriptor in `WEB-INF/web.xml`, as described in the Servlet 2.3 spec and in "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd".

A `taglib` element has two subelements: `taglib-uri` and `taglib-location`.

<taglib>

A taglib is a subelement of web-app:

```
<!ELEMENT web-app .... taglib* .... >
```

The taglib element provides information on a tag library that is used by a JSP page within the Web Application.

A taglib element has two subelements and one attribute:

```
<!ELEMENT taglib ( taglib-uri, taglib-location ) >
<!ATTLIST taglib id ID #IMPLIED>
```

<taglib-uri>

A `taglib-uri` element describes a URI identifying a Tag Library used in the Web Application.

```
<!ELEMENT taglib-uri (#PCDATA) >
PCDATA ::= a URI spec. It may be either an absolute URI
specification, or a relative URI as in Section 2.2.1.
```

<taglib-location>

A `taglib-location` contains the location (as a resource) where to find the Tag Library Description File for this Tag Library.

```
<!ELEMENT taglib-location (#PCDATA) >
PCDATA ::= a resource location, as indicated in Section 2.2.1,
where to find the Tag Library Descriptor file.
```

Example

The use of relative URI specifications enables very short names in the `taglib` directive. For example:

```
<%@ taglib uri="/myPRlibrary" prefix="x" %>
```

and then

```
<taglib>
  <taglib-uri>/myPRlibrary</taglib-uri>
  <taglib-location>/WEB-INF/tlds/PRlibrary_1_4.tld</taglib-uri>
</taglib>
```

9.3.3 Determining the TLD Resource Path

We next describe how to determine the TLD resource path from the uri attribute of a taglib directive.

9.3.3.1 Definitions

An "absolute URI" is one that starts with a protocol and host. 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.

9.3.3.2 Processing WEB.XML.

The web.xml for the web application may contain one or more <taglib></taglib> elements. All such elements are considered. The result of "processing" web.xml is, per each taglib element, two values, a TAGLIB_URI and a TAGLIB_LOCATION, as follows:

For each <taglib> element:

1 The value of the <taglib-uri> subelement is the TAGLIB_URI. This TAGLIB_URI may be an absolute URI, or a relative URI spec starting with "/" or one not starting with "/".

2a If the <taglib-location> subelement is some relative URI specification that starts with a "/" the TAGLIB_LOCATION is this URI.

2b 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 "/").

9.3.3.3 Computing the TLD Resource Path

We describe how to resolve a taglib directive to compute the TLD resource path. We do this based on the value of the uri attribute of the taglib directive. In the description below, ABS_URI stands for an absolute URI, ROOT_REL_URI for a relative URI that starts with "/ ", and NOROOT_REL_URI for a relative URI that does not start with "/".

If uri="ABS_URI":

Look in the processed web.xml for a taglib entry whose TAGLIB_URI is ABS_URI. If found, the corresponding TABLIB_LOCATION is the TLD resource path. If not found, a translation error is raised.

If uri="ROOT_REL_URI":

Look in the processed web.xml for a taglib entry whose TAGLIB_URI is ROOT_REL_URI. If found, the TABLIB_LOCATION for the taglib entry is the TLD resource path. If no such entry is found, ROOT_REL_URI is the TLD resource path.

If uri="NOROOT_REL_URI":

Look in the processed web.xml for a taglib entry whose TAGLIB_URI is NOROOT_REL_URI. If found, the TABLIB_LOCATION for the taglib entry 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. Let ROOT_REL_URI be the resolved value (this is a relative URI specification that starts with "/" - by definition-). ROOT_REL_URI is the URL resource path.

9.3.3.4 Examples

The web.xml map allows very explicit description of the tag libraries that are being used in a Web Application.

The default 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="/tlds/PRlibrary_1_4.tld" prefix="x" %>
```

9.3.4 Translation-Time Class Loader

The set of classes available at translation time is the same as 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 through the use of the `class-path` attribute in the `META-INF/MANIFEST` file of these JAR files.

9.3.5 Assembling a Web Application

As part of the process of assembling a Web Application together, 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 then create a `WEB-INF/web.xml` that ties everything together.

Tag libraries that have been delivered in the standard format can be dropped directly into `WEB-INF/lib`. 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 9.6.3.

9.3.6 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 same 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.

If there is no mapping for a given URI and the URI is not well-known to the JSP container, a translation-time error will occur.

There is no guarantee that this “well-known URI” mechanism will be preserved in later releases of the JSP specification. As experience accumulates on how to use tag extensions, the JSP specification may incorporate new functionality that will make the “well-known URI” mechanism unnecessary; at that point it may be removed.

9.4 The Tag Library Descriptor Format

This section describes the DTD for the Tag Library Descriptor. This is the same DTD as "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd", except for some formatting changes to extract comments and make them more readable.

TLDs in the 1.1 format must be accepted by JSP 1.2 containers.

Notation

```
<!NOTATION WEB-JSPTAGLIB.1_1 PUBLIC “-//Sun Microsystems, Inc.//DTD
JSP Tag Library 1.2//EN”>
```

<taglib>

The **taglib** element is the document root. A taglib has two attributes.

```
<!ATTLIST taglib
     id
       ID
       #IMPLIED
     xmlns
       CDATA
       #FIXED
       "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd"
>
```

A taglib element also has several subelements that define:

tlibversion the version of the tag library implementation

jspversion the version of JSP specification the tag library depends upon

shortname a simple default short name that could be used by a JSP page authoring tool to create names with a mnemonic value; for example, the it may be used as the preferred prefix value in taglib directives.

uri a uri uniquely identifying this taglib.

display-name The display-name element contains a short name that is intended to be displayed by tools.

small-icon Optional large-icon that can be used by tools.

large-icon Optional large-icon that can be used by tools.

info a string describing the “use” of this taglib.

validatorclass Optional TagLibraryValidator class.

listener Optional event listener specification

```
<!ELEMENT taglib
       (tlibversion, jspversion?,
        shortname, uri?, display-name?, small-icon?, large-icon?
        info?, validatorclass?, listener*, tag+) >
```

<tlibversion>

Describes this version (number) of the taglibrary.

The syntax is:

```
<!ELEMENT tlibversion (#PCDATA) >
#PCDATA ::= [0-9]*{ “.”[0-9] }0..3
```

<jspversion>

Describes the JSP specification version (number) this taglibrary requires in order to function. The default is 1.1

The syntax is:

```
<!ELEMENT jspversion  (#PCDATA) >
#PCDATA ::= [0-9]*{ “.”[0-9] }0..3.
```

<shortname>

Defines a simple default short name that could be used by a JSP page authoring tool to create names with a mnemonic value; for example, the it may be used as the preferred prefix value in taglib directives and/or to create prefixes for IDs . Do not use white space, and do not start with digits or underscore.

The syntax is

```
<!ELEMENT shortname      (#PCDATA) >
#PCDATA ::= NMTOKEN
```

<uri>

Defines a public URI that uniquely identifies this version of the tag library. It is recommended that the URI identifying a tag library is actually a URL to the tag library descriptor for this specific version of the tag library.

```
<!ELEMENT uri         (#PCDATA) >
```

<info>

Defines an arbitrary text string describing the tag library.

```
<!ELEMENT info        (#PCDATA) >
```

<validatorclass>

Defines an optional TagLibraryValidator class that can be used to validate the conformance of a JSP page to using this tag library.

```
<!ELEMENT validatorClass        (#PCDATA) >
```

<listener>

Defines an optional event listener object to be instantiated and registered automatically.

```
<!ELEMENT listener        (listener-class) >
```

<listener-class>

The listener-class element declares a class in the application that must be registered as a web application listener bean. See the Servlet 2.3 specification for details.

```
<!ELEMENT listener-class        (#PCDATA) >
```

<tag>

The **tag** defines an action in this tag library. It has one attribute:

```
<!ATTLIST tag id ID #IMPLIED >
```

The **tag** may have several subelements defining:

name	the unique action name
tagclass	the tag handler class implementing `javax.servlet.jsp.tagext.Tag`
teiclass	an optional subclass of `javax.servlet.jsp.tagext.TagExtraInfo`
bodycontent	the body content type
display-name	A short name that is intended to be displayed by tools.
small-icon	Optional large-icon that can be used by tools.
large-icon	Optional large-icon that can be used by tools.
info	Optional tag-specific information.
variable	Optional scripting variable information.

attribute all attributes of this action

The element syntax is as follows:

```
<!ELEMENT tag
       (name, tagclass, teiclass?,
       bodycontent?, display-name?, small-icon?, large-icon?,
       info?, variable*, attribute*) >
```

<tagclass>

Defines the tag handler class implementing the `javax.serlvet.jsp.tagext.Tag` interface. This element is required.

The syntax is:

```
<!ELEMENT tagclass (#PCDATA) >
#PCDATA ::= fully qualified Java class name.
```

<teiclass>

Defines the subclass of `javax.servlet.jsp.tagext.TagExtraInfo` for this tag. This element is optional.

The syntax is:

```
<!ELEMENT teiclass (#PCDATA) >
 #PCDATA ::= fully qualified Java class name
```

<bodycontent>

Provides a hint as to the content of the body of this action. Primarily intended for use by page composition tools.

There are currently three values specified:

tagdependent The body of the action is passed verbatim to be interpreted by the tag handler itself, and is most likely in a different “language”, e.g. embedded SQL statements. The body of the action may be empty.

JSP The body of the action contains elements using the JSP syntax. The body of the action may be empty.

empty The body must be empty

The default value is “JSP”.

The syntax is:

```
<!ELEMENT bodycontent (#PCDATA) >
```

```
#PCDATA ::=  tagdependent | JSP | empty.
Values are case dependent.
```

<display-name>

The display-name elements contains a short name that is intended to be displayed by tools.

The syntax is:

```
<!ELEMENT display-name (#PCDATA) >
```

<large-icon>

The large-icon element contains the name of a file containing a large (32 x 32) icon image. The file name is 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. The icon can be used by tools.

The syntax is:

```
<!ELEMENT large-icon (#PCDATA) >
```

<small-icon>

The small-icon element contains the name of a file containing a small (16 x 16) icon image. The file name is 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. The icon can be used by tools.

The syntax is:

```
<!ELEMENT small-icon (#PCDATA) >
```

<variable>

Provides information on the scripting variables defined by 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 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 required.
class	name of the class of the variable. java.lang.String is default.
declare	whether the variable is declared or not. True is thedefault.

scope the scope of the scripting variable defined. NESTED is default.

The syntax is:

```
<!ELEMENT variable
       ((name-given | name-from-attribute), class?,
       declare?, scope?) >
```

<name-given>

The name for the scripting variable. One of name-given or name-from-attribute is required.

The syntax is:

```
<!ELEMENT name-given (#PCDATA) >
```

<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.

The syntax is:

```
<!ELEMENT name-from-attribute (#PCDATA) >
```

<class>

The optional name of the class for the scripting variable. The default is java.lang.String.

The syntax is:

```
<!ELEMENT class (#PCDATA) >
```

<declare>

Whether the scripting variable is to be defined or not. See TagExtraInfo for details. This element is optional and “true” is the default.

The syntax is:

```
<!ELEMENT declare #PCDATA) >
#PCDATA ::= true | false | yes | no
```

<scope>

The scope of the scripting variable. See TagExtraInfo for details. This element is optional and “NESTED” is the default..

The syntax is:

```
<!ELEMENT scope #PCDATA) >
#PCDATA ::= NESTED | AT_BEGIN | AT_END
```

<*attribute*>

Provides information on an attribute of this action. Attribute defines an id attribute for external linkage.

```
<!ATTLIST attribute id     ID#IMPLIED>
```

The subelements of attribute are of the form:

name	the attributes name (required)
required	if the attribute is required or optional (optional)
rtexprvalue	if the attributes value may be dynamically calculated at runtime by a scriptlet expression (optional)
type	the type of the attributes value (optional)

The syntax is:

```
<!ELEMENT attribute
       (name, required?,
       rtexprvalue?, type?) >
```

<*name*>

Defines the canonical name of a tag or attribute being defined

The syntax is:

```
<!ELEMENT name        (#PCDATA) >
#PCDATA ::= NMTOKEN
```

<*required*>

Defines if the nesting attribute is required or optional.

The syntax is:

```
<!ELEMENT required    (#PCDATA) >
#PCDATA ::= true | false | yes | no
```

If not present then the default is “false”, i.e the attribute is optional.

<rtexprvalue>

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.

The syntax is:

```
<!ELEMENT rtexprvalue (#PCDATA) >
#PCDATA ::= true | false | yes | no
```

If not present then the default is “false”, i.e the attribute has a static value

<type>

Defines the Java type of the attributes value. For static values (those determined at translation time) the type is always `java.lang.String`.

If the rtexprvalue element is true, that is the value of the nesting attribute may be calculated from an expression scriptlet during request processing then the type defines the return type expected from any scriptlet expression specified as the value of this attribute.

The syntax is:

```
<!ELEMENT type        (#PCDATA) >
#PCDATA ::= fully qualified Java class name of result type
```

An example is:

```
<type>
 java.lang.Object
</type>
```

9.5 Validation

There are a number of reasons why the structure of a JSP page should conform to some validation rules. Some of them are:

- Request-time semantics; e.g. a subelement requires at request-time the information from some enclosing element.
- Authoring-tool support; e.g. some tool may require some ordering in the actions.
- Methodological constraints; e.g. a development group may want to constraint 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 1.2 specification provides a very flexible translation-time validation mechanism.

9.5.1 Translation-Time Mechanisms

Some translation-time validation is represented in the Tag Library Descriptor. In some cases a TagExtraInfo class needs to be written to supplement this information.

9.5.1.1 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 `bodycontent` attribute 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.

9.5.1.2 Validator Classes

A `TagLibraryValidator` class may be listed in the TLD for a tag library to request that a JSP page be validated. The JSP page is exposed as its associated XML document through a `PageInfo` class, and the validator class can do any checks the tag library author may have found appropriate.

The validator class mechanism was introduced in the JSP 1.2 specification. We expect that validator classes will be written based on different XML schema mechanisms (DTDs, XSchema, Relaxx, others). A validator class for it may be incorporated into a later version of the JSP specification if a clear schema standard appears at some point.

9.5.1.3 Syntactic Information in a TagExtraInfo Class

Additional translation-time validation can be done using the `isValid` method in the `TagExtraInfo` class. The `isValid` method is invoked at translation-time and is passed a `TagData` instance as its argument.

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 are probably best using the validator class mechanism.

9.5.2 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 `JspException` can be thrown. If uncaught, this object will invoke the errorpage mechanism of the JSP specification.

9.6 Conventions and Other Issues

This section is not normative, although it reflects good design practices.

9.6.1 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; this action will define the desired objects.

Then the JSP page can make these objects available as follows:

```
<%@ tablig 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 some implementation dependency in making these objects available; for example, they may be providing access to some functionality that only exists in some 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, but that is a different discussion).

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.

9.6.2 Access to Vendor-Specific information

If a vendor wants to associate with some tag library some information that is not described in the current version of the TLD, it can do so by inserting the information in a document it controls, inserting the document in the WEB-INF portion of the JAR file where the Tab Library resides, and using the standard Servlet 2.2 mechanisms to access that information.

The vendor can now use the ID machinery to refer to the element within the TLD.

9.6.3 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 2.2 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 10

Tag Extension API

Custom actions can be used by JSP authors and authoring tools to simplify writing JSP pages.

A custom action has a start tag, possibly a body, and an end tag. A prototypical example is of the form:

```
<x:foo att="myObject" >
  BODY
</x:foo/>
```

An empty tag has no body, in which case the start and end tags can be combined as follows:

```
<x:foo att="myObject" />
```

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

Chapter 9 details the how TLDs are used in taglib directives and the format of the TLD file. This chapter describes the methods that are available to access the TLD and the details of the tag extension 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.

10.1 Simple Tag Handlers

In this section we introduce the notion of a tag handler and describe the simplest type of tag handler.

Tag Handler

A tag handler is a special type of run-time server-side object that is created to help evaluate custom actions during the execution of a JSP page. A tag handler is an invisible server-side JavaBeans component that supports an additional protocol for better integration within a JSP page.

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.

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 must be initialized through the appropriate setter methods before the instance can be used. It is the responsibility of the JSP container to invoke the appropriate setter methods to initialize these properties.

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 needs not set it again. The tag handler-specific properties can be reset using the *resetCustomAttributes* method, if it exists.

Conversions

Attribute values in a JSP page are described as strings, but the corresponding properties of the tag handler may be have other types. The following conversions are done.

When the attribute is a request-time attribute value, no conversion is done, i.e. types must match exactly.

When the attribute is a literal string, if there is a PropertyEditor associated with the JavaBean component, then, the setAsText() method will be used. Otherwise, the rules in Table 2-4, section 2.13.2 will be followed.

The Tag Interface

A Tag handler that does not want to process its body can implement just the Tag interface. There are several reasons why a tag handler will not want to process its body: because it has none (there is a mechanism in the TLD to require the JSP parser to verify that), 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.

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.

10.1.1 Tag

Syntax

```
public interface Tag
```

All Known Subinterfaces: BodyTag, IterationTag

Description

The interface of a simple 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().

The JSP 1.2 specification has the resetCustomProperties() method to reset all custom properties to default values. Note that the JSP translator can determine whether a specific tag handler class supports or not this method.

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).

Release

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 end of the page.

Lifecycle details are collected elsewhere in the JSP specification document.

10.1.1.1 Fields

`public static final int` **`EVAL_BODY_INCLUDE`**

Evaluate body into existing out stream. Valid return value for doStartTag. This is an illegal return value for doStartTag when the class implements BodyTag, since BodyTag implies the creation of a new BodyContent.

`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.

10.1.1.2 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 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 and the request is completed. If this request was forwarded or included from another page (or Servlet), only the current page evaluation is completed.

The JSP container will resynchronize any variable values that are indicated as so in TagExtraInfo after the invocation of doEndBody().

Throws:
`JspException.`, JspException

`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 any variable values that are indicated as so in TagExtraInfo after the invocation of doStartBody().

Throws:
`JspException.`, JspException

See Also: BodyTag

`public Tag` **`getParent`**`()`

Get the parent (closest enclosing tag handler) for this tag handler. This method is used by the findAncestorWithClass() method in TagSupport.

Parameters:
`t` - The enclosing tag handler.

`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` **`resetCustomAttributes`**`()`

Reset all custom (i.e. not parent, not pageContext) attributes to their default values

`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

`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. Code can assume that setPageContext has been called with the proper values before this point.

Parameters:
`t` - The parent tag, or null.

10.1.2 IterationTag

Syntax

```
public interface IterationTag extends Tag
```

All Known Subinterfaces: BodyTag

All Superinterfaces: 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.

10.1.2.1 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,

10.1.2.2 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, the value of out will be restored using the popBody method in pageContext, and then doEndTag will be invoked.

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 any variable values that are indicated as so in TagExtraInfo after the invocation of doAfterBody().

Returns: whether additional evaluations of the body are desired

Throws:
JspException

10.1.3 TagSupport

Syntax

```
public class TagSupport implements IterationTag, java.io.Serializable
```

Direct Known Subclasses: BodyTagSupport

All Implemented Interfaces: IterationTag, java.io.Serializable, Tag

Description

A base class for defnining 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 tags.

10.1.3.1 Fields

`protected java.lang.String` **`id`**

`protected PageContext` **`pageContext`**

10.1.3.2 Constructors

`public` **`TagSupport`**`()`

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.

10.1.3.3 Methods

`public int` **`doAfterBody`**`()`

Default processing for a body

Returns: SKIP_BODY

Throws:
JspException

See Also: public int doAfterBody()

`public int` **`doEndTag`**`()`

Default processing of the end tag returning EVAL_PAGE.

Throws:
JspException

See Also: public int doEndTag()

`public int` **`doStartTag`**`()`

Default processing of the start tag, returning SKIP_BODY.

Throws:

JspException

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.

Parameters:
`from` - The instance from where to start looking.

`klass` - The subclass of Tag or interface to be matched

`public java.lang.String` **`getId`**`()`

The value of the id attribute of this tag; or null.

`public Tag` **`getParent`**`()`

The Tag instance most closely enclosing this tag instance.

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.

`public java.util.Enumeration` **`getValues`**`()`

Enumerate the values kept by this tag handler.

`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:
`pageContenxt` - 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.

10.2 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 Tag, provides three additional methods `setBodyContent(BodyContent)`, `doInitBody()` and `doAfterBody()` 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_TAG. 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_TAG, then the body will be evaluated, and doAfterBody() will be invoked again.

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).

10.2.1 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.

10.2.1.1 Constructors

`protected` **`BodyContent`**`(JspWriter e)`

Protected constructor. Unbounded buffer, no autoflushing.

10.2.1.2 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`

`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`

10.2.2 BodyTag

Syntax

```
public interface BodyTag extends IterationTag
```

All Superinterfaces: IterationTag, 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 to capture the body evaluation. This object is obtained by calling the pushBody method of the current pageContext, which additionally has the effect of saving the previous out value. The object is returned through a call to 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.

The new property is 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.

The new action methods is doInitBody(), which is invoked right after setBodyContent() and before the body evaluation. This method is only invoked if doStartTag() returns EVAL_BODY_BUFFERED.

10.2.2.1 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.

10.2.2.2 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. The method will not be invoked if there is no body evaluation.

The JSP container will resynchronize any variable values that are indicated as so in TagExtraInfo after the invocation of doInitBody().

Throws:
JspException

`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. The method will be invoked before doInitBody and it will not be invoked if there is no body evaluation (for example if doStartTag() returns EVAL_BODY_INCLUDE or SKIP_BODY).

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

10.2.3 BodyTagSupport

Syntax

```
public class BodyTagSupport extends TagSupport implements BodyTag
```

All Implemented Interfaces: BodyTag, IterationTag, 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 TagSupport and only redefine a few tags.

10.2.3.1 Fields

`protected BodyContent` **`bodyContent`**

10.2.3.2 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.

10.2.3.3 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

`public int` **`doEndTag`**`()`

Default processing of the end tag returning EVAL_PAGE.

Overrides: public int doEndTag() in class TagSupport

Returns: EVAL_PAGE

Throws:
JspException

`public void doInitBody()`

Prepare for evaluation of the body just before the first body evaluation: no action.

Throws:
JspException

`public int doStartTag()`

Default processing of the start tag returning EVAL_BODY_TAG.

Overrides: public int doStartTag() in class TagSupport

Returns: EVAL_BODY_TAG;

Throws:
JspException

`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

`public void setBodyContent(BodyContent b)`

Prepare for evaluation of the body: stash the bodyContent away.

Parameters:
b - the BodyContent

10.3 Tag Life Cycle

At execution time the implementation of a JSP page will use an available Tag instance with the appropriate prefix and name that is not being used, initialize it, and then follow the protocol

described below. Afterwards, it will release the instance and make it available for further use. This approach reduces the number of instances that are needed at a time.

Initialization is done by setting the properties pageContext and parent, in that order.

Once a tag handler instance has initialized the pageContext and parent properties, all custom properties as indicated through the attributes in the custom action instance, if any, will be set to the values as requested, following any conversions applicable.

Custom properties may have some default values. A tag handler should set its propertes set to any default values it may expect after the pageContext and parent properties are set. Also custom properties may be reset to their default values using the (new) method resetCustom-Attributes(); the values of the standard properties pageContext and parent are preserved by this method.

All properties will be reset to an undetermined state when release() is invoked.

Unset Attributes and Tag Handlers: Reusing Instances

Consider the following JSP fragment:

```
<x:foo att1=“one” att2=“two”/>
<x:foo att1=“HELLO” att2=“BYE”/>
```

To implement this fragment, the JSP page implementation object can use one or two tag handler instances. If it wants to reuse the first tag handler, it just needs to do a h.set-Att1(“HELLO”); h.setAtt2(“BYE”); to prepare the handler for the second action.

Consider now the case of:

```
<foo:bar attr1=“abc” attr2=“def”/>
<foo:bar attr1=“xyz”/>
```

To implement this fragment, the JSP page implementation object can use two tag handler instances, one instance on which the setter methods for &quotattr1“ and &quotattr2” are invoked, and a separate instance for when only the setter for &quotattr1“ is used.

If the tag handler supports the *resetCustomAttributes* method, then a single tag handler instance can also be used, provided that this method is invoked to reset the properties, and then the setter method for &quotattr1“ is used.

If the tag handler does not support the *resetCustomAttributes* method, a *reset* invocation could be done, but then the parent and pageContext properties will have to be reset too.

A Run-Time Trace

The following figure shows the run-time trace for a complex Tag instance; methods invoked by the JSP page code that almost never redefined by a specific Tag handler are in blue, and the action methods are in red.

```
ATag h = new ATag();
...
h.setPageContext(pageContext);
h.setParent(parent);
h.setAttribute1(value1);
h.setAttribute2(value2);
h.doStartTag()
__________
out = pageContext.pushBody()
h.setBodyContent(out)
h.doInitBody()
[BODY]
h.doAfterBody()
.....
[BODY]
h.doAfterBody()
......
out = pageContext.popBody()
__________
h.doEndTag()
h.release()
```

10.4 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 fragment 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 fragment 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 `BodyTagSupport.findAncestorWith-Class(Tag, Class)`, which uses a reference to parent tag kept by each Tag instance, which effectively provides a run-time execution stack.

10.5 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 prefix, name, PageContext, parent, and TagData and then follow the protocol described below. 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.

See the Tag Extensions Chapter of the JSP 1.2 specification for more details.

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 &quotid“ 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.

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 `PageInfo` object, which abstracts the XML view of the JSP page.

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.

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.

Next the XML view of the page is validated according to the validator classes (if any) in all the tag libraries that were used in the JSP page. The view will be exposed to the validator classes as an instance of a PageInfo class. This class will provides an InputStream (read-only) on the page; later specifications may add other views on the page (DOM, SAX, JDOM are all candidates).

The validators are invoked by iterating over all taglib directives in the page, in the order in which they appear:

If the TLD has a <validatorClass> object then
•get an instance of the validator class (container may recycle if wanted)
•set the TagLibraryInfo object on the instance the first time.
•invoke the validate method on the instance.
•report any errors found.

After checking all the tag library validator classes, the TagExtraInfo classes for all tags will be consulted by invoking their `isValid` method. The order of invocation of this methods is undefined.

10.5.1 TagLibraryInfo

Syntax

```
public abstract class TagLibraryInfo
```

Description

Information available at translation-time on a Tag Library. This class is instantiated from the Tag Library Descriptor file (TLD).

10.5.1.1 Fields

```
protected java.lang.String info
protected java.lang.String jspversion
protected java.lang.String prefix
protected java.lang.String shortname
protected TagInfo[] tags
protected java.lang.String tlibversion
protected java.lang.String uri
protected java.lang.String urn
```

10.5.1.2 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

10.5.1.3 Methods

```
public java.lang.String getInfoString()
```

Information (documentation) for this TLD.

`public java.lang.String` **`getPrefixString`**`()`

The prefix assigned to this taglib from the <%taglib directive

`public java.lang.String` **`getReliableURN`**`()`

The “reliable” URN indicated in the TLD. This may be used by authoring tools as a global identifier (the uri attribute) to use when creating an include directive for this library.

`public java.lang.String` **`getRequiredVersion`**`()`

A string describing the required version of the JSP container.

`public java.lang.String` **`getShortName`**`()`

The prefered short name (prefix) as indicated in the TLD. This may be used by authoring tools as the prefered prefix to use when creating an include directive for this 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

`public TagInfo[]` **`getTags`**`()`

An array describing the tags that are defined in this tag library.

`public java.lang.String` **`getURI`**`()`

The value of the uri attribute from the <%@ taglib directive for this library.

10.5.2 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.

10.5.2.1 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_TAG_DEPENDENT`**

static constant for getBodyContent() when it is Tag dependent

10.5.2.2 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. 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:

`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;

10.5.2.3 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. A null return means no attributes.

Returns: The array of TagAttributeInfo for this tag.

`public java.lang.String` **`getBodyContent`**`()`

The bodycontent information for this tag.

Returns: the body content string.

`public java.lang.String` **`getInfoString`**`()`

The information string for the tag.

Returns: the info string

`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 tab library instance we belong to.

`public java.lang.String` **`getTagName`**`()`

The name of the Tag.

Returns: The (short) name of the tag.

`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.

Default is null if the tag has no “id” attribute, otherwise, {“id”, Object}

Parameters:
`data` - TagData describing this action.

Returns: Array of VariableInfo elements.

`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 java.lang.String` **`toString`**`()`

Stringify for debug purposes...

Overrides: java.lang.Object.toString() in class java.lang.Object

10.5.3 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.

10.5.3.1 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.

10.5.3.2 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 hold a request-time Attribute

10.5.3.3 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` **`isRequired`**`()`

Whether this attribute is required.

Returns: if the attribute is required.

`public java.lang.String` **`toString`**`()`

Overrides: java.lang.Object.toString() in class java.lang.Object

10.5.4 PageInfo

Syntax

```
public abstract class PageInfo
```

Description

Translation-time information on a JSP page. The information corresponds to the XML document associated with the JSP page.

Objects of this type are generated by the JSP translator, e.g. when being pased to a TagLibrary-Validator instance.

10.5.4.1 Constructors

```
public PageInfo()
```

10.5.4.2 Methods

```
public abstract java.io.InputStream getInputStream()
```

Returns an input stream on the XML document.

Returns: An input stream on the document.

10.5.5 TagLibraryValidator

Syntax

```
public abstract class TagLibraryValidator
```

Description

Translation-time validator class for a JSP page. A validator operates on the XML document associated with the JSP page.

Validator classes are associated with a tag library via the TLD. A TagLibraryValidator instance is associated with a given TLD and the JSP translator will invoke the setTagLibraryInfo method on an instance before invoking the validate method. 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.

10.5.5.1 Constructors

```
public TagLibraryValidator()
```

10.5.5.2 Methods

```
public TagLibraryInfo getTagLibraryInfo()
```

Get the TagLibraryInfo associated with with Validator.

Returns: The TagLibraryInfo instance

```
public void setTagLibraryInfo(TagLibraryInfo tld)
```

Set the TagLibraryInfo data for this validator.

Parameters:
`tld` - The TagLibraryInfo instance

```
public java.lang.String validate(PageInfo thePage)
```

Validate a JSP page. This method will return a null String if the page passed through is valid; otherwise an error message.

Parameters:
`thePage` - the JSP page object

Returns: A string indicating whether the page is valid or not.

10.5.6 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 must be used:
•if the tag defines any scripting variables
•if the tag wants to provide 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 get-TagInfo().

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 isValid() and before any invocation on getVariableInfo().

10.5.6.1 Constructors

`public` **`TagExtraInfo`**`()`

10.5.6.2 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.

`public boolean` **`isValid`**`(TagData data)`

Translation-time validation of the attributes. Request-time attributes are indicated as such in the TagData parameter.

Parameters:
`data` - The TagData instance.

Returns: Whether this tag instance is valid.

`public final void` **`setTagInfo`**`(TagInfo tagInfo)`

Set the TagInfo for this class.

Parameters:
`tagInfo` - The TagInfo this instance is extending

10.5.7 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 and getVariableInfo methods of TagExtra-Info, which are invoked at translation time.

10.5.7.1 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).

10.5.7.2 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.

10.5.7.3 Methods

`public java.lang.Object` **`getAttribute`**`(java.lang.String attName)`

The value of the attribute. Returns the distinguished object REQUEST_TIME_VALUE if the value is request time. Returns null if the attribute is not set.

Returns: the attribute's value object

`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.

Returns: the attribute value string

`public java.lang.String` **`getId`**`()`

The value of the id attribute, if available.

Returns: the value of the id attribute or null

```
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.

10.5.8 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 may have scope page, request, session, and application.

The class name (VariableInfo.getClassName) in the returned objects are used to determine the types of the scripting variables. Because of this, a custom action cannot create a scripting variable of a primitive type. The workaround is to use "boxed" types.

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 2.3 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 (will check if there is a JLS verbiage to refer to). The class must also be in the CLASSPATH for the Web Application (see Servlet 2.3 specification - essentially it is WEB-INF/lib and WEB-INF/classes). Failure to be so will lead to a translation-time error.

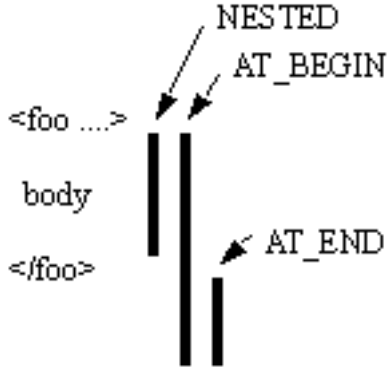
Usage Comments

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 almost other remaining cases it will refer to a class that is in the platform on which the JSP processor is build (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 defined values for scope are:

- 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 page.
- AT_END, if the scripting variable is available after the end tag of the action that defines it until the end of the page.

The scope value for a variable implies what methods may affect its value and thus, in lack of additional information, where synchronization is needed:

- for NESTED, after doInitBody and doAfterBody for a tag handler implementing BodyTag, and after doStartTag otherwise.
- for AT_BEGIN, after doInitBody, doAfterBody, and doEndTag for a tag handler implementing BodyTag, and doStartTag and doEndTag otherwise.
- for AT_END, after doEndTag method.

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.

10.5.8.1 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

10.5.8.2 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:
`id` - The name of the scripting variable

`className` - The name of the scripting 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

10.5.8.3 Methods

```
public java.lang.String getClassName()
public boolean getDeclare()
public int getScope()
public java.lang.String getVarName()
```

APPENDIX 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 2.3 and JSP 1.2 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 2.3 capabilities to the JSP 1.2 capabilities. The appendix is included here as this is a feature that JSP page authors and JSP page authoring tools are interested in.

A.1 Backward Compatibility Note

Note – We will clarify under what conditions a JSP 1.2 page can be compiled into a Servlet that can run on a Servlet 2.2 container. At the present, the only issue that seems important is that described in Issue # 17 of Errata 1.1_a, *"PageContext and Handling Throwable or Exception"*.

A.2 A very simple JSP page

We start with a very simple JSP page `HelloWorld.jsp`.

```
<%@ page info="Example JSP pre-compiled" %>
<p>
Hello World
</p>
```

A.3 The 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 webappSYSTEM "http://java.sun.com/j2ee/dtds/web-app_1_2.dtd">
<webapp>
        <session-config>
                <session-timeout> 1 </session-timeout>
        </session-config>
</webapp>
```

A.4 The 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 `_jsp_HelloWorld_XXX_Impl`. The Servlet code only depends on the JSP 1.2 and Servlet 2.3 APIs, as follows:

```
imports javax.servlet.*;
imports javax.servlet.http.*;
imports javax.servlet.jsp.*;

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);
                    }
            }
}
```

A.5 The 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_1_2.dtd">
<webapp>
       <servlet>
                  <servlet-name> HelloWorld </servlet-name>
                  <servlet-class> HelloWorld.class </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>
```

A.6 The WAR for the compiled JSP page

Finally everything is packaged together into a WAR:

`/WEB-INF/web.xml`

`/WEB-INF/classes/HelloWorld.class`

Note that if the Servlet class generated for the JSP page had dependent on some support classes, they would have to be included in the WAR.

APPENDIX B

Changes

This appendix lists the changes in the JavaServer Pages specification.

B.1 Changes between 1.1 and 1.2 PD1

The following changes ocurred between the JSP 1.1 and JSP 1.2 Public Draft 1.

B.1.1 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.

B.1.2 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.

B.1.3 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.

B.1.4 Clarifications

- Incorporated errata 1.1_a and (in progress) 1.1_b.

B.1.5 Changes

- JSP 1.2 is based on Servlet 2.3, in particular:
- JSP 1.2 is based on the Java 2 platform.

B.2 Changes 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.

B.2.1 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 C.
- Expanded Chapter 1 so as to cover 0.92’s "model 1" and "model 2".
- Clarified a number of JSP 1.0 details.

B.2.2 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 C

Glossary

This appendix is a glossary of the main concepts mentioned in this specification.

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.

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 An 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.

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.

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

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`.

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 file A text file that contains a JSP page. In the current version of the specification, the JSP file must have a *.jsp* extension.

JSP page A text-based document that uses fixed template data and JSP elements and describes how to process a *request* to create a *response*. The semantics of a JSP page are realized at runtime by a JSP page implementation class.

JSP page, front A 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.

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".

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.

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 handler A Java class that implements the Tag or the BodyTag interfaces and that is the run-time representation of a custom action.

tag handler A JavaBean component that implements the `Tag` or `BodyTag` interfaces and is the run-time 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.