JavaServer Pages™ Specification

Version 1.2

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

Monday, August 27, 2001

Eduardo Pelegrí-Llopart, editor

Sun microsystems
We're the dot in .com™

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: FCS
Release: September 17, 2001

NOTICE

RESTRICTED RIGHTS LEGEND

REPORT

Contents

Status

This is the JSP 1.2 specification, developed by the expert group JSR053 under the Java Community Process (more details at http://jcp.org/jsr/detail/53.jsp).

JSP.S.1 The Java Community Process

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

JSP.S.2 The JCP and this Specification

The JCP is designed to be a very flexible process so each expert group can address the requirements of the specific communities it serves. The reference implementation for JSP 1.2 and Servlet 2.3 uses code that is being developed as an open source project under an agreement with the Apache Software Foundation.

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

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

Appendices C and D are normative; the other appendices are non-normative.

Preface

This document is the JavaServer Pages™ 1.2 Specification (JSP 1.2).

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

JSP.P.1 Relation To JSP 1.1

JSP 1.2 extends JavaServer Pages™ 1.1 Specification (JSP 1.1) in the following ways:

- Requiring the Java 2 platform, version 1.2 or later.
- Using Servlet 2.3 as the foundation for its semantics.
- Defining the XML syntax for JSP pages
- Providing for translation-time validation of JSP pages.
- Specifying refinements of tag library runtime support.
- Improving the tag handler contract.
- Providing improved support for page authoring.
- Improving character encoding and localization support.
- Fixing the infamous “flush before you include” limitation in JSP 1.1.

JSP.P.2 Licensing of Specification

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

JSP.P.3 Who should read this document

This document is the authoritative JSP 1.2 specification. It is intended to provide requirements for implementations of JSP processing, and support by web containers in web servers and application servers.

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

JSP.P.4 Related Documents

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

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

JSP home page	**http://java.sun.com/products/jsp**
Servlet home page	**http://java.sun.com/products/servlet**
Java 2 Platform, Standard Edition	http://java.sun.com/products/jdk/1.3
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

JSP.P.5 Historical Note

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

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

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

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

JSP.P.6 Acknowledgments

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

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

We want to thank:

Alex Yiu, Alex Chaffee, Allan Scott, Amit Kishnani, Bill dehOra, Bjorn Carlson, Bob Foster, Chris Hansen, Clement Wong, Craig McClanahan, Dano Ferrin, Danny Coward, Dave Brown, Edwin Smith, Francios Jouaux, Frank Biederich, Govind Seshadri, Hans Bergsten, Howard Melman, James Strachan, Jason McGeee, Jason Hunter, Jeff Mischkinsky, Jon Rousseau, Julie Basu, Karl Avedal, Kevin Jones, Larry Cable, Larry Isaas, Magnus Rydin, Magnus Stenman, Mark Wallace, Miles Sabin, Misha Davidson, Murty Chintalapati, Nathan

Abamson, Nic Ferrier, Olli Blackburn, Paul Bonfanti, Peter Petersen, Petr Jiricka, Pier Paolo Fumagalli, Pierre Delisle, Ramesh Mandava, Rod Magnuson, Sam Pullara, Satoshi Kikuchi, Scott Ferguson, Scott Snyder, Simeon Simeonov, Stephanie Bodoff, Stefano Mazzocchi, Tim Ampe, Tom Reilly, Vince Bonfanti.

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

Special mention is due to Scott Ferguson, Bob Foster, Stefano Mazzocchi and Ricardo Rocha for their contributions to chapter 5. Bob Foster deserves credit for for the DTD & XSchema, and the Cocoon community for Cocoon & XSP. Our thanks to Jess Holle for insisting on the topics in Chapter JSP.3.

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

The editor wants to give special thanks to many individuals within the Java Platform Group, and especially to James, Jon, Mala, Jeff, Connie and Graham.

Lastly, 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 JSP.1 Overview

This chapter provides an overview of the JavaServer Pages technology.

JSP.1.1 The JavaServer Pages™ Technology

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

JSP.1.1.1 General Concepts

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

- *Template Data*

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

- *Addition of Dynamic Data*

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

- *Encapsulation of Functionality*

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

- *Good Tool Support*

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

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

JSP.1.1.2 Benefits of the JavaServer Pages Technology

JavaServer Pages technology offers the following benefits:

- *Write Once, Run Anywhere*™ *properties*

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

- *High quality tool support*

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

- *Separation of Roles*

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

- *Reuse of components and tag libraries*

  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 with interactive tools for component development and page composition, yielding considerable development time savings. In addition, they provide the cross-platform power and flexibility of the Java programming language or other scripting languages.

- *Separation of dynamic and static content*

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

- *Support for scripting and actions*

  The JavaServer Pages technology supports scripting elements as well as actions. Actions *encapsulate* useful functionality in a convenient form that can 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). The J2EE platform brings Java technology to enterprise computing. You can now develop powerful middle-tier server applications that include a web site using JavaServer Pages technology as a front end to Enterprise JavaBeans components in a J2EE compliant environment.

JSP.1.2 Basic Concepts

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

JSP.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 dynamic actions and leverages on the Java 2 Platform. JSP technology supports a number of different paradigms for authoring dynamic content. The key features of JavaServer Pages are:

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

JSP.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) running in client(s) (downloadable via the Plugin and Java Web Start technology)

The JavaServer Pages specification inherits from the Servlet specification the concepts of web applications, ServletContexts, sessions, requests and responses. See the Java Servlet 2.3 specification for more details.

JSP.1.2.3 Components and Containers

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

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

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

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

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

JSP.1.2.5 Role in the Java 2 Platform, Enterprise Edition

With a few exceptions, integration of JSP pages within the J2EE 1.3 platform is inherited from the Servlet 2.3 specification since translation turns JSPs into servlets.

CHAPTER JSP.2

Core Syntax and Semantics

This chapter describes the core syntax and semantics for the JavaServer Pages 1.2 specification (JSP 1.2).

JSP.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. The processing of the JSP page may involve creating and/or using other objects.

A JSP page defines a *JSP page implementation class* that implements the semantics of the JSP page. This class is a subclass of Servlet (see Chapter JSP.8 for details). At request time a request intended for the JSP page is delivered to the JSP page implementation object for processing.

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

JSP.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 with JSP container.

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

JSP.2.1.2 XML Document for a JSP Page

JSP pages have an equivalent XML document. The XML view of a JSP page is exposed to the translation phase (see below).

A JSP page can be written directly as an XML document. Beginning with JSP 1.2, the XML document can be delivered to a JSP container for processing.

It is not valid to mix standard syntax and XML syntax in the same source file. However, a JSP page in either syntax can include a JSP page in either syntax via a directive.

JSP.2.1.3 Translation and Execution Phases

A JSP container manages two phases of a JSP page's life. In the translation phase, the container determines a JSP page implementation class that corresponds to the JSP page. In the execution phase the container manages one or more instances of this class in response to requests and other events.

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

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

During the *execution phase* the JSP container delivers events to the JSP page implementation object. The container is responsible for instantiating request and response objects and invoking the appropriate JSP page implementation object. Upon completion of processing, the response object is received by the container for communication to the client. The details of the contract between the JSP page implementation class and the JSP container are described in Chapter JSP.8.

The translation of a JSP source page into its implementation class can occur at any time between initial deployment of the JSP page into the JSP container and the receipt and processing of a client request for the target JSP page.
Section JSP.2.1.5 describes how to perform the translation phase ahead of deployment.

JSP.2.1.4 Events in JSP Pages

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

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

JSP.2.1.5 Compiling JSP Pages

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

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

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

A JSP page can also be compiled at deployment time.

JSP.2.1.5.1 JSP Page Packaging

When a JSP page implementation class depends on support classes (in addition to the JSP 1.2 and Servlet 2.3 classes), the support classes are included in the packaged WAR (as defined in the Servlet 2.3 specification) for portability across JSP containers..

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

1. A JSP page delivered in source form (probably the most common case).
1. A JSP page translated into an implementation class plus deployment information. The deployment information indicates support classes needed and the mapping between the original URL path to the JSP page and the URL for the JSP page implementation class for that page.

JSP.2.1.6 Debugging JSP Pages

In the past debugging tools provided by development environments have lacked a standard format for conveying source map information allowing the debugger of one vender to be used with the JSP container of another. A specification for debugging support that overcomes this limitations is being worked on under JSR-045 of the JCP 2.0 process with the title “Debugging Support for Non-Java Languages”. Details can be obtained at http://jcp.org/jsr/detail/45.jsp.

JSP.2.1.7 Naming Conventions for JSP Files

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

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

The Servlet 2.3 specification wires-in the extension “.jsp” to mean a JSP page, but does not differentiate top JSP files from included fragments. We recommend, but do not mandate, that:

- “.jsp” files correspond to top level JSP files containing a JSP page.
- Included fragments not use the “.jsp” extension. Any other extension will do, although “.jspf” and “.jsf” seem reasonable extensions and are offered as suggestions.

JSP.2.2 Web Applications

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

- Java runtime environment(s) running in the server (required)
- JSP page(s) that handle requests and generate dynamic content
- Servlet(s) that handle requests and generate dynamic content

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

Web applications are described in more detail in the Servlet 2.3 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 these resources be implicitly associated with and accessible through a unique ServletContext instance available as the implicit application object (Section JSP.2.8).

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

- The include directive (Section JSP.2.10.3).
- The jsp:include action element (Section JSP.4.4).
- The jsp:forward action (Section JSP.4.5).

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.

JSP.2.2.1 Relative URL Specifications

Elements may use *relative URL specifications*, called "URI paths" in the Servlet 2.3 specification. These paths are as described in the RFC 2396 specification. We refer to the path part of that specification, not the scheme nor authority parts. Some examples are:

A *context-relative* path is a path that starts with a "/". It is to be interpreted as relative to the application to which the JSP page belongs, that is to say that its ServletContext object provides the base context URL.

A *page relative* path is a path that does not start with a "/". It is to be interpreted as relative to the current JSP page or the current JSP file depending on where the path is being used: for an include directive (Section JSP.2.10.3) where the path is used in a file attribute, the interpretation is relative to the JSP file; for an jsp:include action (Section JSP.4.4) where the path is used in a page attribute, the interpretation is relative to the JSP page. In both cases the current page (or file) is denoted by some path starting with "/" that is then modified by the new specification to produce a path starting with "/". The new path is interpreted through the ServletContext object. See Section JSP.2.10.4 for exact details on this interpretation.

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

JSP.2.3 Syntactic Elements of a JSP Page

This section describes the basic syntax rules of JSP pages.

JSP.2.3.1 Elements and Template Data

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

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

JSP.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

Actions provide information for the request processing phase. The interpretation of an *action* may, and often will, depend on the details of the specific request received by the JSP page. An Actions can either be *standard*, that is. defined in this specification, or *custom*, that is provided via the portable tag extension mechanism.
Action elements follow the syntax of an XML element.: They have a start tag including the element name, and may have attributes, an optional body, and a matching end tag, or they be an empty tag possibly with attributes:

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

and

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

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

Scripting Elements

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

JSP.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. The start tag cannot be on one file while the end tag is in another.

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

A scripting language may also impose constraints on the placement of start and end tags relative to specific scripting constructs. For example, Chapter 6 shows that Java language blocks cannot separate a start and an end tag; see Section JSP.6.4 for details.

JSP.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

As examples, the following are all empty tags:

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

While the following are all non-empty tags:

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

JSP.2.3.5 Attribute Values

Following the XML specification, attribute values always appear quoted. Either single or double quotes can be used to reduce the need for quoting quotes; the quotation conventions available are described in Section JSP.2.6. There are two types of attribute values, literals and request-time expressions (Section JSP.2.13.1) but the quotation rules are the same.

JSP.2.3.6 Valid Names for Actions and Attributes

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

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

JSP.2.3.7 White Space

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

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

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

Table 2.1: Example 1 - Input

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

The result is

Table 2.2: Example 1 - Output

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

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

Table 2.3: Example 2 - Input

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

The result is

Table 2.4: Example 2 - Output

LineNo	Output Text
1	<?xml version=”1.0” ?>
2	

Table 2.4: Example 2 - Output

4	The rest of the document goes here

JSP.2.4 Error Handling

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

JSP.2.4.1 Translation Time Processing Errors

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

JSP.2.4.2 Request Time Processing Errors

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

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

Any uncaught exceptions thrown in the body of the JSP page implementation class result in the forwarding of the client request and uncaught exception to the

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

errorPage URL specified by the 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 setAttribute() 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 reserved and 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

JSP.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.

JSP.2.5.1 Generating Comments in Output to Client

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

```
<!-- comments ... -->
```

These comments are treated as uninterpreted template text by the JSP container. If the generated comment is to have dynamic data, this can be obtained through an expression syntax, as in:

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

JSP.2.5.2 JSP Comments

A JSP comment is of the form

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

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

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

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

JSP.2.6 Quoting and Escape Conventions

The following quoting conventions apply to JSP pages.

Quoting in Scripting Elements

- A literal %> is quoted by %\>

Quoting in Template Text

- A literal <% is quoted by <\%

Quoting in Attributes

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

- A ‘ is quoted as \’. This is required within a single quote-delimited attribute value.
- A “ is quoted as \”. This is required within a double quote-delimited attribute value.
- A \ is quoted as \\
- A %> is quoted as %\>
- A <% is quoted as <\%
- The entities &apos; and &quot; are available to describe single and double quotes.

Examples

The following line shows an illegal attribute values.

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

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

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

The next lines are all legal quotations.

- <%= "Joe said %/>" %>
- <%= "Joe said %\>" %>
- <% String joes_statement = "hi!"; %>
  <%= "Joe said \"" + joes_statement + "\"." %>
  <x:tag value='<%="Joe said \\"" + joes_statement + "\\"."%>'/>
- <x:tag value='<%= "hi!" %>' />
- <x:tag value="<%= \"hi!\" %>" />
- <x:tag value='<%= \"name\" %>' />
- <x:tag value="<%= \"Joe said 'hello'\" %>"/>
- <x:tag value="<%= \"Joe said \\\"hello\\\" \" %>"/>
- <x:tag value="end expression %\>"/>
- <% String s="abc"; %>
  <x:tag value="<%= s + \"def\" + \"jkl\" + 'm' + \'n\' %>" />
  <x:tag value='<%= s + \"def\" + "jkl" + \'m\' + \'n\' %>' />

XML Representation

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

JSP.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 JSP.6). Most details are not language specific and are described in this chapter.

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

- Initially, *out* is a new JspWriter object. This object may be different from the stream object returned from response.getWriter(), and may be considered to be interposed on the latter in order to implement buffering (see Section JSP.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.
- The JSP container should not invoke response.getWriter() until the time when the first portion of the content is to be sent to the client. This enables a number of uses of JSP, including using JSP as a language to ‘glue’ actions that deliver binary content, or reliably forwarding to a servlet, or change dynamically the content type of the respose before generating content. See Chapter JSP.3.
- Within the body of some actions, *out* may be temporarily re-assigned to a different (nested) instance of JspWriter object. Whether this is the case depends on the details of the action’s semantics. Typically the content of these temporary streams is appended to the stream previously referred to by *out*, and *out* is subsequently re-assigned to refer to the previous (nesting) stream. Such nested streams are always buffered, and require explicit flushing to a nesting stream or their contents will be discarded.
- If the *initial out* JspWriter object is buffered, then depending upon the value of the autoFlush attribute of the page directive, the content of that buffer will either be automatically flushed out to the ServletResponse output stream to obviate overflow, or an exception shall be thrown to signal buffer overflow. If the *initial out* JspWriter is unbuffered, then content written to it will be passed directly through to the ServletResponse output stream.

A JSP page can also describe what should happen when some specific *events* occur. In JSP 1.2, the only events that can be described are the initialization and the destruction of the page. These events are described using “well-known method names” in declaration elements. (See Section JSP.8.1.1.1).

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

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

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

JSP.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 may exist depending on the *scope* of the object.

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

The exact rules for the visibility of the variables are scripting language specific. Chapter JSP.2.1 defines the rules for when the language attribute of the `page` directive is “java”.

JSP.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 JSP.2.8.3, “Implicit Objects). Other objects are created explicitly through actions, or created directly using scripting code. Created objects have a *scope attribute* defining *where* there is a reference to the object and *when* that reference is removed.

The created objects may also be visible directly to scripting elements through scripting-level variables (see Section JSP.2.8.3, “Implicit Objects).

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

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

- *page* - Objects with *page* scope are accessible only within the page where they are created. All references to such an object shall be released after the response is sent back to the client from the JSP page or the request is forwarded somewhere else. References to objects with *page* scope are stored in the pageContext object.
- *request* - Objects with *request* scope are accessible from pages processing the same request where they were created. References to the object shall be released after the request is processed. In particular, if the request is forwarded to a resource in the same runtime, the object is still reachable. References to objects with *request* scope are stored in the request object.
- *session* - Objects with *session* scope are accessible from pages processing requests that are in the same session as the one in which they were created. It is not legal to define an object with session scope from within a page that is not session-aware (see Section JSP.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. Objects with application scope can be defined (and reached) from pages that are not session-aware. References to objects with *application* scope are stored in the application object associated with a page activation. The application object is the servlet context obtained from the servlet configuration object. All references to the object shall be released when the runtime environment reclaims the ServletContext.

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

JSP.2.8.3 Implicit Objects

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

Each implicit object has a class or interface type defined in a core Java technology or Java Servlet API package, as shown in **Table JSP.2-1**.

Table JSP.2-1 *Implicit Objects Available in JSP Pages*

Variable Name	Type	Semantics & Scope
request	protocol dependent subtype of: javax.servlet.ServletRequest e.g: javax.servlet.http.HttpServletRequest	The request triggering the service invocation. **request** scope.
response	protocol dependent subtype of: javax.servlet.ServletResponse, e.g: javax.servlet.http.HttpServletResponse	The response to the request. **page** scope.
pageContext	javax.servlet.jsp.PageContext	The page context for this JSP page. **page** scope.
session	javax.servlet.http.HttpSession	The session object created for the requesting client (if any). This variable is only valid for Http protocols. **session** scope
application	javax.servlet.ServletContext	The servlet context obtained from the servlet configuration object (as in the call getServletConfig(). getContext()) **application** scope
out	javax.servlet.jsp.JspWriter	An object that writes into the output stream. **page** scope

Table JSP.2-1 *Implicit Objects Available in JSP Pages*

Variable Name	Type	Semantics & Scope
config	javax.servlet.ServletConfig	The `ServletConfig` for this JSP page **page** scope
page	java.lang.Object	The instance of this page's implementation class processing the current request[a] **page** scope

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

In addition, the exception implicit object can be accessed in an error page, as described in Table JSP.2-2.

Table JSP.2-2 *Implicit Objects Available in Error Pages*

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

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

See Section JSP.7.6.1 for some non-normative conventions for the introduction of new implicit objects.

JSP.2.8.4 The pageContext Object

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

See Chapter JSP.9 for more details.

JSP.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 JSP.2.6, “Quoting and Escape Conventions.

JSP.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 JSP.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.

JSP.2.10.1 The page Directive

The page directive defines a number of page dependent properties 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 16KB should be used.

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

Syntax

```
<%@ page page_directive_attr_list %>

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

The details of the attributes are as follows:

Table JSP.2-1

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 JSP.6. All scripting languages must provide some implicit objects that a JSP page author can use in declarations, scriptlets, and expressions. The specific objects that can be used are defined in Section JSP.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 JSP.8). This attribute should not be used without careful consideration as it restricts the ability of the JSP container to provide specialized superclasses that may improve on the quality of rendered service. See Section JSP.7.6.1 for an alternate way to introduce objects into a JSP page that does not have this drawback.

Table JSP.2-1

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 in that package. The import list shall be imported by the translated JSP page implementation and is thus available to the scripting environment. The default import list is java.lang.*, javax.servlet.*, javax.servlet.jsp.* and javax.servlet.http.*. This 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 then output is buffered with a buffer size not less than that specified. Depending upon the value of the "autoFlush" attribute, the contents of this buffer is either automatically flushed, or an exception is raised, when overflow would occur. The default is buffered with an implementation buffer size of not less than `8kb`.

Table JSP.2-1

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 the shared state of the page. Default is "true". Note that even if the *isThreadSafe* attribute is "false" the JSP page author must ensure that accesses to any shared objects are properly synchronized., The objects may be shared in either the ServletContext or the HttpSession**.**
info	Defines an arbitrary string that is incorporated into the translated page, that can subsequently be obtained from the page's implementation of Servlet.getServletInfo() method.
isErrorPage	Indicates if the current JSP page is intended to be the URL target of another JSP page's errorPage. If "true", then the implicit script language variable "exception" is defined and its value is a reference to the offending Throwable from the source JSP page in error. If "false" then the "exception" implicit variable is unavailable, and any reference to it within the body of the JSP page is illegal and shall result in a fatal translation error. Default is "false"

Table JSP.2-1

errorPage	Defines a URL to a resource to which any Java programming language Throwable object(s) thrown but not caught by the page implementation are forwarded for error processing. The provided URL spec is as in Section JSP.2.2.1. If the URL names another JSP page then, when invoked that JSP page's exception implicit script variable shall contain a reference to the originating uncaught Throwable. The default URL is implementation dependent. Note the Throwable object is transferred by the throwing page implementation to the error page implementation by saving the object reference on the common ServletRequest object using the setAttribute() method, with a name of "javax.servlet.jsp.jspException". 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 Chapter 3 for complete details on character encodings.
pageEncoding	Defines the character encoding for the JSP page. Values is of the form "CHARSET" which must be the IANA value for a character encoding. The CHARSET value of contentType is used as default if present, or ISO-8859-1 otherwise. See Chapter 3 for complete details on character encodings.

JSP.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.

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

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

Examples

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

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

Syntax

<%@ taglib uri=”*tagLibraryURI*” prefix=”*tagPrefix*” %>

where the attributes are:

Table JSP.2-1

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

Table JSP.2-1

tagPrefix	Defines the *prefix* string in `<prefix>:<tagname>` that is used to distinguish a custom action, e.g <myPrefix:myTag>. Prefixes starting with jsp:, jspx:, java:, javax:, servlet:, sun:, and sunw: are reserved. A prefix must follow the naming convention specified in the XML namespaces specification. Empty prefixes are illegal in this version of the specification.

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

JSP.2.10.3 The include Directive

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

Examples

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

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

Syntax

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

JSP.2.10.4 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 JSP.2-1.

Table JSP.2-1 *Summary of Include Mechanisms in JSP 1.2*

Syntax	Spec	Object	Description	Section
Include Directive - Translation-time				
<%@ include file=... %>	file-relative	static	Content is parsed by JSP container.	JSP.2.10.3
Include Action - Request-time				
<jsp:include page= />	page-relative	static and dynamic	Content is not parsed; it is included in place.	JSP.4.4

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

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

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

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

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

The precise definition of the semantics for scripting done using elements based on the Java programming language is given in Chapter JSP.6.

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

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

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

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

The equivalent XML elements for these scripting elements are described in Section JSP.5.2.

JSP.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, global to the page. The second declaration does the same and initializes it to zero. This type of initialization should be done with care in the presence of multiple requests on the page. The third declaration declares a method global to the page.

```
<%! int i; %>

<%! int i = 0; %>

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

Syntax

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

JSP.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 JSP.6).

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

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

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

Examples

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

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

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

```
<% int i; i= 0; %>
Hi, the value of i is <% i++ %>
```

Syntax

```
<% scriptlet %>
```

JSP.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. The result is subsequently emitted into the current out JspWriter object.

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

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

The expression must be a complete expression in the scripting language in which it is written.

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

This example inserts the current date.

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

Syntax

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

JSP.2.12 Actions

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

The JSP specification includes some actions that are *standard* and must be implemented by all conforming JSP containers; these actions are described in Chapter 4.

New actions are defined according to the mechanisms described in Chapters 7 and 10 and are introduced using the taglib directive.

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

JSP.2.13 Tag Attribute Interpretation Semantics

The interpretation of all actions start by evaluating the values given to its attributes left to right, and assigning the values to the attributes. In the process some conversions may be applicable; the rules for them are described in Section JSP.2.13.2.

Many values are fixed 'translation-time values', but JSP 1.2 also provides a mechanism for describing values that are computed at request time, the rules are described in Section JSP.2.13.1.

JSP.2.13.1 Request Time Attribute Values

An attribute value of the form "<%= scriptlet_expr %>" or '<%= scriptlet_expr %>' denotes a *request-time attribute value*. The value denoted is that of the scriptlet expression involved. Request-time attribute values can only be used in actions and cannot be used in directives. If there are more than one such attribute in a tag, the expressions are evaluated left-to-right.

Quotation is done as in any other attribute value (Section JSP.2.6).

Only attribute values can be denoted this way (the name of the attribute is always an explicit name). The expression must appear by itself (multiple expressions, and mixing of expressions and string constants are not permitted). Multiple operations must be performed within the expression. Type conversions are described in Section JSP.2.13.2.

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

Most attributes in the standard actions from Chapter 4 have page translation-time semantics, but the following attributes accept request-time attribute expressions:

- The value attribute of jsp:setProperty (Section JSP.4.2).
- The beanName attribute of jsp:useBean (Section JSP.4.1).
- The page attribute of jsp:include (Section JSP.4.4).
- The page attribute of jsp:forward (Section JSP.4.5).
- The value attribute of jsp:param (Section JSP.4.6).
- The height and width attributes of jsp:plugin (Section JSP.4.7).

JSP.2.13.2 Type Conversions

We describe two cases for type conversions

JSP.2.13.2.1 Conversions from String values

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

String values can be used to assign values to a type that has a PropertyEditor class as indicated in the JavaBeans specification. When that is the case, the setAsText(String) method is used. A conversion failure arises if the method throws an IllegalArgumentException.

String values can also be used to assign to the types as listed in Table JSP.2-2. The conversion applied is that shown in the table.

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

Table JSP.2-2 *Conversions from string values to target type*

Target Type	Source String Value
Bean Property	Use setAsText(string-literal)
boolean or Boolean	As indicated in java.lang.Boolean.valueOf(String)

Table JSP.2-2 *Conversions from string values to target type*

byte or Byte	As indicated in java.lang.Byte.valueOf(String)
char or Character	As indicated in String.charAt(0)
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)
short of Short	As indicated in java.lang.Short.valueOf(String)
Object	As if new String(string-literal)

These conversions are part of the generic mechanism used to assign values to attributes of actions: when an attribute value that is not a request-time attribute is assigned to a given attribute, the conversion described here is used, using the type of the attribute as the target type. The type of each attribute of the standard actions is described in this specification, while the types of the attributes of a custom action are described in its associated Tag Library Descriptor.

A given action may also define additional ways where type/value conversions are used. In particular, Section JSP.4.2 describes the mechanism used for the setProperty standard action.

JSP.2.13.2.3 Conversions from request-time expressions

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

CHAPTER JSP.3

Localization Issues

This chapter describes requirements for localization with JavaServer Pages 1.2 (JSP 1.2).

JSP.3.1 Page Character Encoding

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.

A 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

A JSP page uses a character encoding. The encoding can be described explicitly using the pageEncoding attribute of the page directive. The character encoding defaults to the encoding indicated in the contentType attribute of the page directive if it is given, or to ISO-8859-1 otherwise. ISO-8859-1 is also known as latin-1.

The valid names for describing character encodings in JSP 1.2 are those of IANA. They are described at:

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

The pageEncoding attribute should be used only when the character encoding of a JSP page is organized so that ASCII characters stand for themselves. The directive containing the pageEncoding attribute should appear as early as possible in the JSP page.

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

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

JSP.3.2 Static Content Type

Most JSP pages are written to deliver a response using a specific content type and character encoding. A JSP page can use the contentType attribute of the page directive to indicate the content type of the response it provides to requests.

When used this way, 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 forward the request to the other JSP page or servlet.

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

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

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

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.

JSP.3.3 Dynamic Content Type

Some JSP pages are designed so they can deliver content using different content types and character encodings depending on request time input. These pages may be organized as custom actions or scriptlets that determine the response content type and provide 'glue' into other code actually generating the content of the response.

Dynamic setting of content type relies on an underlying invocation on response.setContentType(). That method can be invoked as long as no content has been been sent to the response stream. Data is sent to the response stream on buffer flushes for buffered pages, or on encountering the first content (beware of whitespace) on unbuffered pages.

Whitespace is notoriously tricky for JSP pages in JSP syntax, but much more manageable for JSP pages in XML syntax.

JSP.3.4 Delivering Localized Content

The JSP specification does not mandate any specific approach for structuring localized content, and different approaches are possible. Two common approaches are to use a template taglib and pull localized strings from a resource repository, or to use-per-locale JSP pages. Each approach has benefits and drawbacks. Some users have been using transformations on JSP documents to do simple replacement of elements by localized strings, thus maintaining JSP syntax with no performance cost at run-time. Combinations of these approaches also make sense.

There are a number of different efforts that are exploring how to best do localization. We expect JSR-052, the standard JSP tag library, to address some of these issues.

CHAPTER JSP.4

Standard Actions

This chapter describes the standard actions of JavaServer Pages 1.2 (JSP 1.2).

JSP.4.1 <jsp:useBean>

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

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

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

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

The attribute beanName specifies the name of a Bean, as specified in the JavaBeans specification. It is used as an argument to the instantiate() method in the java.beans.Beans class. It must be of the form "a.b.c", which may be either a class, or the name of a resource of the form "a/b/c.ser" that will be resolved in the current ClassLoader. If this is not true, a request-time exception, as indicated in the semantics of instantiate() will be raised. The value of this attribute can be a request-time attribute expression.

More detail on the role of id and scope is given next.

The id Attribute

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

- the *name* must be unique within the translation unit, and identifies the particular element in which it appears to the JSP container and page.
  Duplicate id’s found in the same translation unit shall result in a fatal translation error.

- The JSP container will associate an object (a JavaBean component) with the named value and accessed via that name in various contexts through the *pagecontext* object described later in this specification.
  The *name* is also used to expose a variable (name) in the page’s scripting language environment. The scope of the scripting language variable is dependent upon the scoping rules and capabilities of the scripting language used in the page.
  Note that this implies the *name* value syntax must comply with the variable naming syntax rules of the scripting language used in the page. Chapter JSP.6 provides details for the case where the language attribute is ”java”.

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

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

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

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

The ***scope*** *Attribute*

The scope=”page|request|session|application” attribute/value tuple is associated with, and modifies the behavior of the id attribute described above (it has both translation time and client request processing time semantics). In particular it describes the namespace, the implicit lifecycle of the object reference associated with the *name*, and the APIs used to access this association, as follows:

Table JSP.4-1

page	The named object is available from the javax.servlet.jsp.PageContext for the current page. This reference must be discarded upon completion of the current request by the page body. It is illegal to change the instance object associated so that its runtime type is a subset of the type of the current object previously associated.

Table JSP.4-1

request	The named object is available from the current page's ServletRequest object using the getAttribute(name) method. This reference must be discarded upon completion of the current client request. It is illegal to change the value of an instance object so associated so 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 getAttribute(name) method. This reference must be discarded upon invalidation of the current session. It is Illegal to change the value of an instance object so associated so 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.

Semantics

The actions performed in a jsp:useBean action are:

1. An attempt to locate an object based on the attribute values id and scope. The inspection is done synchronized per scope namespace to avoid non-deterministic behavior.
1. A scripting language variable of the specified type (if given) or class (if type is not given) is defined with the given id in the current lexical scope of the scripting language.
2. If the object is found, the variable's value is initialized with a reference to the

located object, cast to the specified type. If the cast fails, a java.lang.ClassCastException shall occur. This completes the processing of this jsp:useBean action.

3. If the jsp:useBean element had a non-empty body it is ignored. This completes the processing of this jsp: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 jsp: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 the class is instantiated. The new object reference is associated with the scripting variable and with the specified name in the specified scope using the appropriate scope dependent association mechanism (see PageContext). After this, step 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 jsp: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. Any template text will be passed through to the out stream. Scriptlets and action tags will be evaluated.

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

Examples

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

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

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

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

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

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

Syntax

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

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

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

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

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

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

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

Table JSP.4-1

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

JSP.4.2 <jsp:setProperty>

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

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

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

When assigning from a parameter in the request object, the conversions described in Section JSP.2.13.2.1 are applied, using the target property to determine the target type.

When assigning from a value given as a String constant, the conversions described in Section JSP.2.13.2.1 are applied, using the target property to determine the target type.

When assigning from a value given as a request-time attribute, no type conversions are applied, as indicated in Section JSP.2.13.2.3.

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, whether at translation time or request-time.

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 %>” />
```

Syntax

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

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

propertyValue ::= *string*

The value *propertyValue* can also be a request-time attribute value, as described in Section JSP.2.13.1.

propertyValue ::= expr_scriptlet[1]

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

Table JSP.4-2

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.
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.

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

Table JSP.4-2

value	The value to assign to the given property. This attribute can accept a request-time attribute expression as a value. An action may not have both param and value attributes.

JSP.4.3 <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 jsp:useBean action).

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

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

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

The object named by the name must have been "introduced" to the JSP processor using either the jsp:useBean action or a custom action with an associated *VariableInfo* entry for this name.

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

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

Examples

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

Syntax

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

The attributes are:

Table JSP.4-3

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

JSP.4.4 <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 JSP.2-1 for a summary of include facilities.

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

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

An included page only has access to the JspWriter object and it cannot set headers. This precludes invoking methods like setCookie(). Attempts to invoke these methods will be ignored. The constraint is equivalent to the one imposed on the include() method of the RequestDispatcher class.

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

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

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

Examples

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

The above example is a simple inclusion of an object. The path is interpreted in the context of the Web Application. It is likely a static object, but it could be mapped into, for instance, a Servlet via web.xml.

For an example of a more complex set of inclusions, consider the following four situations built using four JSP files: A.jsp, C.jsp, dir/B.jsp and dir/C.jsp:

- **A.jsp** says <%@ include file=”dir/B.jsp”%> and **dir/B.jsp** says <%@ include file=”C.jsp”%>. In this case the relative specification “C.jsp” resolves to “dir/C.jsp”
- **A.jsp** says <jsp:include page=”dir/B.jsp”/> and **dir/B.jsp** says <jsp:include page=”C.jsp” />. In this case the relative specification “C.jsp” resolves to “dir/C.jsp”.
- **A.jsp** says <jsp:include page=”dir/B.jsp”/> and **dir/B.jsp** says <%@ include file=”C.jsp” %>. In this case the relative specification “C.jsp” resolves to “dir/C.jsp”.
- **A.jsp** says <%@ include file=”dir/B.jsp”%> and **dir/B.jsp** says <jsp:include page=”C.jsp”/>. In this case the relative specification “C.jsp” resolves to “C.jsp”.

Syntax

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

and

```
<jsp:include page=”urlSpec” flush="true|false">
    { <jsp:param .... /> }*
</jsp:include>
```

The first syntax just does a request-time inclusion. In the second case, the values in the param subelements are used to augment the request for the purposes of the inclusion.

The valid attributes are:

Table JSP.4-4

`page`	The URL is a relative *urlSpec* is as in Section JSP.2.2.1. Relative paths are interpreted relative to the *current* JSP page. Accepts a request-time attribute value (which must evaluate to a String that is a relative URL specification).
`flush`	Optional boolean attribute. If the value is “true”, the buffer is flushed now. The default value is “false”.

JSP.4.5 <jsp:forward>

A `<jsp:forward page=”urlSpec” />` element allows the runtime dispatch of the current request to a static resource, a JSP page 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 JSP.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, the buffer is cleared prior to forwarding.

If the page output is buffered and the buffer was flushed, an attempt to forward the request will result in an `IllegalStateException`.

If the page output was unbuffered and anything has been written to it, an attempt to forward the request will result in an `IllegalStateException`.

Examples

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

```
<% String whereTo = “/templates/”+someValue; %>
<jsp:forward page=’<%= whereTo %>’ />
```

Syntax

<jsp:forward page=”*relativeURLspec*” />

and

```
<jsp:forward page=”urlSpec”>
    { <jsp:param .... /> }*
</jsp:forward>
```

This tag allows the page author to cause the current request processing to be effected by the specified attributes as follows:

Table JSP.4-5

page	The URL is a relative urlSpec is as in Section JSP.2.2.1. Relative paths are interpreted relative to the *current* JSP page. Accepts a request-time attribute value (which must evaluate to a String that is a relative URL specification).

JSP.4.6 <jsp:param>

The jsp:param element is used to provide key/value information. This element is used in the jsp:include, jsp:forward and jsp:params elements. A translation error shall occur if the element is used elsewhere.

When doing jsp:include or jsp:forward, the included page or forwarded page will see the original request object, with the original parameters augmented with the new parameters, with new values taking precedence over existing values when applicable. The scope of the new parameters is the jsp:include or jsp:forward call; i.e. in the case of an jsp:include the new parameters (and values) will not apply after the include. This is the same behavior as in the ServletRequest include and forward methods (see Section 8.1.1 in the Servlet 2.2 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.

JSP.4.6.1 Syntax

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

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

JSP.4.7 <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 need not be bundled with the JSP container and a reference to Sun's plugin location can be used instead, although some vendors will choose to include the plugin for the benefit of their customers.

Examples

```
<jsp:plugin type=applet code=”Molecule.class” codebase=”/html” >
    <jsp:params>
        <jsp:param
            name=”molecule”
            value=”molecules/benzene.mol”/>
    </jsp:params>
    <jsp:fallback>
        <p> unable to start plugin </p>
    </jsp:fallback>
</jsp:plugin>
```

Syntax

```
<jsp: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>
```

Table JSP.4-1

type	Identifies the type of the component; a Bean, or an Applet.
code	As defined by HTML spec
codebase	As defined by HTML spec
align	As defined by HTML spec
archive	As defined by HTML spec
height	As defined by HTML spec. Accepts a run-time expression value.
hspace	As defined by HTML spec.
jreversion	Identifies the spec version number of the JRE the component requires in order to operate; the default is: “1.2”
name	As defined by HTML spec

Table JSP.4-1

vspace	As defined by HTML spec
title	As defined by the HTML spec
width	As defined by HTML spec. Accepts a run-time expression value.
nspluginurl	URL where JRE plugin can be downloaded for Netscape Navigator, default is implementation defined.
iepluginurl	URL where JRE plugin can be downloaded for IE, default is implementation defined.

JSP.4.8 <jsp:params>

The jsp:params action is part of the jsp:plugin action and can only occur as a direct child of a <jsp:plugin> element. Using the jsp:params element in any other context shall result in a translation-time error.

The semantics and syntax of jsp:params are described in Section JSP.4.7.

JSP.4.9 <jsp:fallback>

The jsp:fallback action is part of the jsp:plugin action and can only occur as a direct child of a <jsp:plugin> element. Using the jsp:fallback element in any other context shall result in a translation-time error.

The semantics and syntax of jsp:fallback are described in Section JSP.4.7.

CHAPTER JSP.5

JSP Documents

This chapter defines an XML syntax for JSP pages and the interpretation of the pages written in this syntax. We use the term *JSP document* to refer to a JSP page in XML syntax.

This chapter also defines a mapping between any JSP page and an XML description of the page, its *XML view*. The XML view is defined for JSP pages written in JSP and in XML syntax.

JSP.5.1 Uses for XML Syntax for JSP Pages

The XML syntax for JSP pages can be used in a number of ways, including:

- JSP documents can be passed directly to the JSP container; this will become more important as more and more content is authored as XML.
- The XML view of a JSP page can be used for validating the JSP page against some description of the set of valid pages.
- JSP documents can be manipulated by XML-aware tools.
- A JSP document can be generated from a textual representation by applying an XML transformation, like XSLT.
- A JSP document can be generated automatically, say by serializing some objects

Validation of the JSP page is supported in the JSP 1.2 specification through a TagLibraryValidator class associated with a tag library. The validator class acts on a PageData object that represents the XML view of the JSP page (see, for example, Section JSP.7.5.1.2).

A JSP page in either syntax can include via a directive a JSP page in either syntax. It is not valid, however, to intermix standard JSP syntax and XML syntax inside the same source file.

JSP.5.2 JSP Documents

A JSP document is a namespace-aware XML document. Namespaces are used to identify the core syntax and the tag libraries used in the page and all JSP-related namespaces are introduced in the root of the XML document.

The core syntax is identified through its own URI. Although in this chapter the prefix jsp is used, any prefix is valid as long as the correct URI identifying the core syntax is used.

A JSP document uses the same file extension (.jsp) as a JSP page in JSP syntax. The container can distinguish the two because a JSP document is an XML document with a jsp:root top element, and a jsp:root cannot appear in a JSP page in JSP syntax.

Many XML elements in a JSP document correspond to JSP language elements, but it is possible to include XML elements that describe template directly. Those elements may have qualified names (and thus be in a namespace), or be unqualified.

A JSP page in XML syntax can use the following elements:

- a jsp:root element that is used to introduce the namespace for custom tags in the page.
- JSP directive elements
- JSP scripting elements
- JSP standard action elements
- JSP custom action elements
- jsp:text elements corresponding to template data.
- other XML fragments also corresponding to template data.

JSP.5.2.1 Semantic Model

The semantic model of a JSP document is unchanged from that of a JSP page in JSP syntax: JSP pages generate a response stream of characters from template data and dynamic elements. Template data can be described explicitly through a jsp:text element, or implicitly through an XML fragment. Dynamic elements are scripting elements, standard actions or custom actions. Scripting elements are represented as XML elements with the exception of request-time expressions, which are represented through special attribute syntax.

To clearly explain the processing of whitespace, we follow the structure of the XSLT specification. The first step in processing a JSP document is to identify the nodes of the document. Then, all textual nodes that have only white space are dropped from the document; the only exception are nodes in a jsp:text element, which are kept verbatim. The resulting nodes are interpreted as described in the following sections. Template data is either passed directly to the response or it is mediated through (standard or custom) actions.

Following the XML specification (and the XSLT specification), whitespace characters are #x20, #x9, #xD or #xA..

JSP.5.2.2 The jsp:root element

A JSP document has jsp:root as its root element. The root element has a xmlns attribute that enables the use of the standard elements defined in the JSP 1.2 specification.

In addition, the root is where namespace attributes of taglibs will be inserted. All tag libraries used within the JSP document are represented in the root element through additional xmlns attributes. The root element has one mandatory attribute, the version of the JSP specification the page is using. No other attributes are defined in this element.

```
<jsp:root
    xmlns:jsp=”http://java.sun.com/JSP/Page”
    xmlns:prefix1=”URI-for-taglib1”
    xmlns:prefix2=”URI-for-taglib2”... >
    version="1.2">
        JSP page
</jsp:root>
```

An xmlns attribute for a custom tag library of the form *xml:prefix=’uri’* identifies the tag library through the *uri* value. The *uri* value may be of one of two forms, either a “uri” or of the form “urn:jsptld:*path*”.

If the *uri* value is of the form “urn:jsptld:*path*”, then the TLD is determined following the mechanism described in Section JSP.7.3.2.

If the *uri* value is a plain “uri”, then a path is determined by consulting the mapping indicated in web.xml extended using the implicit maps in the packaged tag libraries (Sections JSP.7.3.3 and JSP.7.3.4), as indicated in Section JSP.7.3.6.

JSP.5.2.3 The jsp:directive.page element

The jsp:directive.page element defines a number of page dependent properties and communicates these to the JSP container. This element must be a child of the root element and appear at the beginning of the JSP document. Its syntax is:

```
<jsp:directive.page page_directive_attr_list />
```

Where page_directive_attr_list is as described in Section JSP.2.10.1.

The interpretation of a jsp:directive.page element is as described in Section JSP.2.10.1, and its scope is the JSP document and any fragments included through an include directive.

JSP.5.2.4 The jsp:directive.include element

The jsp:directive.include element is used to substitute text and/or code at JSP page translation-time. This element can appear anywhere within a JSP document. Its syntax is:

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

The interpretation of a jsp:directive.include element is as in Section JSP.2.10.3.

The XML view of a JSP page does not contain jsp:directive.include elements, rather the included file is expanded in-place. This is done to simplify validation.

JSP.5.2.5 The jsp:declaration element

The jsp:declaration element is used to declare scripting language constructs that are available to all other scripting elements. A jsp:declaration element has no attributes and its body is the declaration itself. Its syntax is:

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

The interpretation of a jsp:declaration element is as in Section JSP.2.11.1.

JSP.5.2.6 The jsp:scriptlet element

The jsp:scriptlet element is used to describe actions to be performed in response to some request. Scriptlets that are program fragments. A jsp:scriptlet element has no attributes and its body is the program fragment that comprises the scriptlet. Its syntax is:

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

The interpretation of a jsp:scriptlet element is as in Section JSP.2.11.2.

JSP.5.2.7 The jsp:expression element

The jsp:expression element is used to describe complete expressions in the scripting language that get evaluated at response time. A jsp:expression element has no attributes and its body is the expression. Its syntax is:

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

The interpretation of a jsp:expression element is as in Section JSP.2.11.3.

JSP.5.2.8 Standard and custom action elements

A JSP document may use the standard actions described in Chapter JSP.4. Since the syntax of those actions is already based on XML, the description in that chapter suffices, except that in a JSP document template text can be described either through a jsp:text element or through an XML element that is neither a standard nor a custom action (see Section JSP.5.2.11). For completeness, the action elements are:

- jsp:useBean
- jsp:setProperty
- jsp:getProperty
- jsp:include
- jsp:forward
- jsp:param
- jsp:params
- jsp:plugin
- jsp:text

The semantics and constraints are as in Chapter JSP.4, and the interpretation of the scripting elements is as in Chapter JSP.6. Tag libraries introduce new elements through an xmlns attribute on a jsp:root element and their syntax and semantics are as in Chapter JSP.7.

JSP.5.2.9 Request-Time Attributes

An action element that can accept a request time attribute (Section JSP.2.13.1) can accept an argument for that attribute of the form “%= *text* %” (white space around *text* is not needed, and note the missing ‘<‘ and ‘>’). The *text*, after any applicable quoting as in any other XML document, is an expression to be evaluated as in Section JSP.2.13.1.

JSP.5.2.10 The jsp:text element

A jsp:text element can be used to enclose template data in the XML representation. A jsp:text element has no attributes and can appear anywhere that template data can. Its syntax is:

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

The interpretation of a jsp:text element is to pass its content through to the current value of out. This is very similar to the XSLT xsl:text element.

JSP.5.2.11 Other XML elements

The XML syntax for JSP pages also allows an XML element that does not represent neither a standard action nor a custom actionto appear anywhere where a jsp:text may appear.

The interpretation of such an XML element is to pass its textual representation to the current value of out, after the whitespace processing described in Section JSP.5.2.1.

As an example, if the relevant fragment of the JSP document is.

Table 5.1: Example 1 - Input

LineNo	Source Text
1	`<hello><jsp:scriptlet>i=3;</jsp:scriptlet>`
2	` <hi>`
3	` <jsp:text> hi you all`
4	` </jsp:text><jsp:expression>i</jsp:expression>`
5	` </hi>`
6	`</hello>`

The result is

Table 5.2: Example 1 - Output

LineNo	Output Text
1	<hello> <hi> hi you all
2	3 </hi></hello>

Note the treatment of whitespace.

JSP.5.3 XML View of a JSP Page

This section describes the XML view of a JSP page: the mapping between a JSP page, written in either XML syntax or in JSP syntax, and an XML document describing it.

JSP.5.3.1 JSP Documents

The XML view of a JSP page written in XML syntax is very close to the original JSP page. Only two transformations are performed:

- Expand all include directives into the JSP fragments they include.
- If the JSP container supports the jsp:id attribute, add the attribute. See Section JSP.5.3.13.

JSP.5.3.2 JSP pages in JSP syntax

The XML view of a JSP page written in XML syntax is defined by the following transformation:

- Expand all include directives into the JSP fragments they include.
- Add a jsp:root element as the root, with appropriate xmlns:jsp attribute, and convert the taglib directive into xmlns: attributes of the jsp:root element.
- Convert declarations, scriptlets and expressions into valid XML elements as described in Section JSP.5.2.2 and following sections.
- Convert request-time attribute expressions as in Section JSP.5.3.11.
- Convert JSP quotations to XML quotations.

- Create jsp:text elements for all template text.
- If the JSP container supports the jsp:id attribute, add the attribute. See Section JSP.5.3.13.

Note that the XML view of a JSP page has no DOCTYPE information; see Section JSP.5.4.

A quick overview of the transformation is shown in Table JSP.5-1:

Table JSP.5-1 *XML view transformations*

JSP page element	XML view
<%-- comment --%>	removed
<%@ page ... %>	<jsp:directive.page ... />. Optionally add jsp:id
<%@ taglib ... %>	jsp:root element is annotated with namespace information. Optionally add jsp:id.
<%@ include ... %>	expanded in place
<%! ... %>	<jsp:declaration> .... </jsp:declaration>. Optionally add jsp:id.
<% ... %>	<jsp:scriptlet> ... </jsp:scriptlet>. Optionally add jsp:id.
<%= ... %>	<jsp:expression> ... </jsp:expression>. Optionally add jsp:id.
Standard action	Replace with XML syntax (adjust request-time expressions; optionally add jsp:id)
Custom action	As is (adjust request-time expressions; optionally add jsp:id)
template	Replace with jsp:text. Optionally add jsp:id.

In more detail:

JSP.5.3.3 JSP comments

JSP comments (of the form *<%-- comment --%>*) are not passed through to the XML view of a JSP page.

JSP.5.3.4 The page directive

A page directive of the form

```
<%@ page { attr=”value” }* %>
```

is translated into an element of the form:

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

JSP.5.3.5 The taglib directive

A taglib directive of the form

<%@ taglib uri=”*uriValue*” prefix=”prefix” %>

is translated into an `xmlns:prefix` attribute on the root of the JSP document, with a value that depends on *uriValue*. If *uriValue* is a relative path, then the value used is “urn:jsptld:*uriValue*”; otherwise, the *uriValue* is used directly.

JSP.5.3.6 The include directive

An include directive of the form

<%@ include file=”*value*” %>

is expanded into the JSP fragment indicated by *value*. This is done to allow for validation of the page.

JSP.5.3.7 Declarations

Declarations are translated into a *jsp:declaration* element. For example, the second example from Section JSP.2.11.1:

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

is translated into the following.

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

Alternatively, we could use an &lt; and instead say:

```
<jsp:declaration> public String f(int i) { if (i&lt;3) return(“...”); } </jsp:declaration>
```

JSP.5.3.8 Scriptlets

Scriptlets are translated into a *jsp:scriptlet* element. In the XML document corresponding to JSP pages, directives are represented using the syntax:

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

JSP.5.3.9 Expressions

In the XML document corresponding to JSP pages, directives are represented using the *jsp:expression* element:

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

JSP.5.3.10 Standard and Custom Actions

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

JSP.5.3.11 Request-Time Attribute Expressions

Request-time attribute expressions are of the form "<%= expression %>". Although this syntax is consistent with the syntax used elsewhere in a JSP page, it is not a legal XML syntax. The XML mapping for these expressions is into values of the form "%= expression' %", where the JSP specification quoting convention has been converted to the XML quoting convention.

JSP.5.3.12 Template Text and XML Elements

All text that is uninterpreted by the JSP translator is converted into the body for a jsp:text element. As a consequence no XML elements of the form described in Section JSP.5.2.11 will appear in the XML view of a JSP page written in JSP syntax.

JSP.5.3.13 The jsp:id Attribute

A JSP container may, optionally, support a jsp:id attribute. This attribute can only be present in the XML view of a JSP page and can be used to improve the quality of translation time error messages. It is optional, and a conforming JSP container may choose not to support it.

In a JSP container that supports the jsp:id attribute, the XML view of any JSP page will have an additional jsp:id attribute added to all XML elements. This attribute is given a value that is unique over all elements in the XML view. See Chapter 10 for more details.

JSP.5.4 Validating an XML View of a JSP page

The XML view of a JSP page is a namespace-aware document and it cannot be validated against a DTD except in the most simple cases. To reduce confusions and possible unintended performance consequences, the XML view of a JSP page will not include a DOCTYPE. Still, since DTDs can have some value as documentation, Appendix JSP.C contains both a DTD and an XSchema description of JSP documents.

There are several mechanisms that are aware of namespaces that can be used to do validation of XML views of JSP pages. The most popular mechanism is the W3C XML Schema language, but others are also suited, including some very simple ones that may check, for example, that only some elements are being used, or, inversely, that they are not used. The TagLibraryValidator for a tag library permits encapsulating this knowledge with a tag library.

The TagLibraryValidator acts on the XML view of the JSP page. If the page was authored in JSP syntax, that view does not provide any detail on template data (all being grouped inside jsp:text elements), but fine detail can be described when using JSP documents[1].

JSP.5.5 Examples

This section presents two examples of JSP documents. The first shows a JSP page in JSP syntax and its mapping to XML syntax. The second shows a JSP page in XML syntax that includes XML fragments.

JSP.5.5.1 A JSP page and its corresponding JSP document

Here is an example of mapping between JSP and XML syntax.

[1.] Similarly, when applying an XSLT transformation to a JSP document, XML fragments will be plainly visible, while the content of jsp:text elements will not

JSP PAGE IN JSP SYNTAX

```
<html>
<title>positiveTagLib</title>
<body>

<%@ taglib uri="http://java.apache.org/tomcat/examples-taglib" prefix="eg" %>
<%@ taglib uri="/tomcat/taglib" prefix="test" %>
<%@ taglib uri="WEB-INF/tlds/my.tld" prefix="temp" %>

<eg:test toBrowser="true" att1="Working">
Positive Test taglib directive </eg:test>
</body>
</html>
```

XML VIEW OF JSP PAGE:

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

<jsp:text><![CDATA[<html>
<title>positiveTagLig</title>
<body>






]]></jsp:text>
<eg:test toBrowser="true" att1="Working>
<jsp:text>Positive test taglib directive</jsp:text>
</eg:test>
<jsp:text><![CDATA[
</body>
</html>
]]></jsp:text>
</jsp:root>
```

JSP.5.5.2 A JSP document

This is an example of a very simple JSP document that has some template XML elements.

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
   xmlns:mytags="prefix1-URL"
   version="1.2">
  <mytags:iterator count="4">
    <foo> </foo>
   </mytags:iterator>
</jsp:root>
```

CHAPTER JSP.6

Scripting

This chapter describes the details of the Scripting Elements when the language directive value is “java”.

The scripting language is based on the Java programming language (as specified by “The Java Language Specification”), but note that there is no valid JSP page, or a subset of a page, that is a valid Java program.

The following sections describe the details of the relationship between the scripting declarations, scriptlets, and scripting expressions, and the Java programming language. The description is in terms of the structure of the JSP page implementation class. A JSP container need not generate the JSP page implementation class, but it must behave as if one exists.

JSP.6.1 Overall Structure

Some details of what makes a JSP page legal are very specific to the scripting language used in the page. This is especially complex since scriptlets are language fragments, not complete language statements.

JSP.6.1.1 Valid JSP Page

A JSP page is valid for a Java Platform if and only if the JSP page implementation class defined by Table JSP.6-1 (after applying all include directives), together with any other classes defined by the JSP container, is a valid program for the given Java Platform, and if it passes the validation methods for all the tag libraries associated with the JSP page.

JSP.6.1.2 Reserved Names

Sun Microsystems reserves all names of the form {_}jsp_* and {_}jspx_*, in any combination of upper and lower case, for the JSP specification. Names of this form that are not defined in this specification are reserved by Sun for future expansion.

JSP.6.1.3 Implementation Flexibility

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

Table JSP.6-1 *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 {

Table JSP.6-1 *Structure of the JavaProgramming Language Class*

(2) Implicit Objects Section	// code that defines and initializes request, response, page, pageContext etc.
(3) Main Section	// code that defines request/response mapping
close of _jspService method	}
close of _jspXXX	}

JSP.6.2 Declarations Section

The declarations section corresponds to the declaration elements.

The contents of this section is determined by concatenating all the declarations in the page in the order in which they appear.

JSP.6.3 Initialization Section

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

JSP.6.4 Main Section

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

The content of code segment 2 is determined from scriptlets, expressions, and the text body of the JSP page. The elements are processed sequentially in the order in which they appear in the page. The translation for each one is determined as indicated below, and its translation is inserted into this section. The translation depends on the element type:

JSP.6.4.1 Template Data

Template data is transformed into code that will place the template data into the stream named by the implicit variable out when the code is executed. White space is preserved.

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

Original	Equivalent Text
template	out.print(template)

JSP.6.4.2 Scriptlets

A *scriptlet* is transformed into its code fragment.:

Original	Equivalent Text
<% fragment %>	fragment

JSP.6.4.3 Expressions

An *expression* is transformed into a Java statement to insert the value of the expression, converted to java.lang.String if needed, into the stream curnamed by the implicit variable `out`. No additional newlines or space is included.

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

Original	Equivalent Text
<%= expression %>	out.print(expression)

JSP.6.4.4 Actions

An action defining one or more objects is transformed into one or more variable declarations for those objects, together with code that initializes the variables. Their visibility is affected by other constructs, for example scriptlets.

The semantics of the action type determines the names of the variables (usually the name of an id attribute, if present) and their type. The only standard action in the JSP specification that defines objects is the jsp:usebean action. The name of the variable introduced is the name of the `id` attribute and its type is the type of the class attribute.

Original	Equivalent Text
<x:tag>	declare AT_BEGIN variables
foo	{
</x:tag>	declare NESTED variables
	transformation of foo
	}
	declare AT_END variables

Note that the value of the scope attribute does not affect the visibility of the variables within the generated program. It affects where and thus for how long there will be additional references to the object denoted by the variabl

CHAPTER JSP.7

Tag Extensions

This chapter describes the tag library facility for introducing new actions into a JSP page. The tag library facility includes portable run-time support, a validation mechanism, and authoring tool support.

This chapter provides an overview of the tag library concept. It describes the Tag Library Descriptor, and the taglib directive. A detailed description of the APIs involved follows in Chapter JSP.10.

JSP.7.1 Introduction

A Tag Library abstracts functionality used by a JSP page by defining a specialized (sub)language that enables a more natural use of that functionality within JSP pages.

The actions introduced by the Tag Library can be used by the JSP page author in JSP pages explicitly, when authoring the page manually, or implicitly, when using an authoring tool. Tag Libraries are particularly useful to authoring tools because they make intent explicit and the parameters expressed in the action instance provide information to the tool.

Actions that are delivered as tag libraries are imported into a JSP page using the `taglib` directive. They are available for use in the page using the prefix given by the directive. An action can create new objects that can be passed to other actions, or can be manipulated programmatically through a scripting element in the JSP page.

Tag libraries are portable: they can be used in any legal JSP page regardless of the scripting language used in that page.

The tag extension mechanism includes information to:

- Execute a JSP page that uses the tag library.
- Author or modify a JSP page.
- Validate the JSP page.
- Present the JSP page to the end user.

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

JSP.7.1.1 Goals

The tag extension mechanism described in this chapter addresses the following goals. It is designed to be:

Portable - An action described in a tag library must be usable in any JSP container.
Simple - Unsophisticated users must be able to understand and use this mechanism. Vendors of JSP functionality must find it easy to make it available to users as actions.
Expressive - The mechanism must support a wide range of actions, including nested actions, scripting elements inside action bodies, and creation, use and updating of scripting variables.
Usable from different scripting languages - Although the JSP specification currently only defines the semantics for scripts in the Java programming language, we want to leave open the possibility of other scripting languages.
Built upon existing concepts and machinery- We do not want to reinvent what exists elsewhere. Also, we want to avoid future conflicts whenever we can predict them.

JSP.7.1.2 Overview

The processing of a JSP page conceptually follows these steps:

Parsing

JSP pages can be authored using two different syntaxes: a JSP syntax and an XML syntax. The semantics and validation of a JSP syntax page is described with reference to the semantics and validation of an equivalent document in the XML syntax.
The first step is to parse the JSP page. The page that is parsed is as expanded by the processing of include directives. Information in the TLD is used in this step, including the identification of custom tags, so there is some processing

of the taglib directives in the JSP page.

Validation

The tag libraries in the XML document are processed in the order in which they appear in the page.
Each library is checked for a validator class. If one is present, the whole document is made available to its validate() method as a PageData object. If the JSP container supports jsp:id, then this information can be used to provide location information on errors.
Each custom tag in the library is checked for a TagExtraInfo class. If one is present, its isValid() method is invoked.

Translation

Finally, the XML document is processed to create a JSP page implementation class. This process may involve creating scripting variables. Each custom action will provide information about variables, either statically in the TLD, or more flexibly by using the getVariableInfo method of a TagExtraInfo class.

Execution

Once a JSP page implementation class has been associated with a JSP page, the class will be treated as any other Servlet class: Requests will be directed to instances of the class. At run-time, tag handler instances will be created and methods will be invoked in them.

JSP.7.1.2.1 Tag Handlers

The semantics of a specific custom action in a tag library is described via a tag handler class which is usually instantiated at runtime by the JSP page implementation class. When the tag library is well known to the JSP container (Section JSP.7.3.9), the container can use alternative implementations as long as the semantics are preserved.

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

The JSP page implementation class instantiates a tag handler object, or reuses an existing tag handler object, for each action in the JSP page. The handler object is a Java object that implements the javax.servlet.jsp.tagext.Tag interface. The handler object is responsible for the interaction between the JSP page and additional server-side objects.

There are three main interfaces: Tag, IterationTag, and BodyTag.

- The Tag interface defines the basic methods needed in all tag handlers. These methods include setter methods to initialize a tag handler with context data and attribute values of the action, and the doStartTag() and doEndTag() methods.
- The IterationTag interface is an extension to Tag that provides the additional method, doAfterBody(), invoked for the reevaluation of the body of the tag.
- The BodyTag interface is an extension of IterationTag with two new methods for when the tag handler wants to manipulate the tag body: setBodyContent() passes a buffer, the BodyContent object, and doInitBody() provides an opportunity to process the buffer before the first evaluation of the body into the buffer.

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

JSP 1.2 has a new interface designed to help maintain data integrity and resource management in the presence of exceptions. The TryCatchFinally interface is a “mix-in” interface that can be added to a class implementing any of Tag, IterationTag or BodyTag.

JSP.7.1.2.2 Event Listeners

A tag library may include classes that are event listeners (see the Servlet 2.3 specification). The listeners classes are listed in the tag library descriptor and the JSP container automatically instantiates them and registers them. A container is required to locate all TLD files (see Section JSP.7.3.1 for details on how they are identified), read their listener elements, and treat the event listeners as extensions of those listed in web.xml.

The order in which the listeners are registered is undefined, but they are registered before application start.

JSP.7.1.3 Simple Examples

As examples, we describe prototypical uses of tag extensions, briefly sketching how they take advantage of these mechanisms.

JSP.7.1.3.1 Simple Actions

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

This type of action can be implemented with a tag handler that implements the Tag interface. The tag handler needs to use only the doStartTag() method which is invoked when the start tag is encountered. It can access the attributes of the tag and information about the state of the JSP page. The information is passed to the Tag object through setter method calls, prior to the call to doStartTag().

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

JSP.7.1.3.2 Actions with a Body

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

JSP.7.1.3.3 Conditionals

In some cases, a body needs to be invoked only when some (possibly complex) condition happens. Again, this type of action is supported by the basic Tag interface through the use of return values in the doStartTag() method.

JSP.7.1.3.4 Iterations

For iteration the IterationTag interface is needed. The doAfterBody() method is invoked to determine whether to reevaluate the body or not.

JSP.7.1.3.5 Actions that Process their Body

Consider an action that evaluates its body many times, creating a stream of response data. The IterationTag protocol is used for this.

If the result of the reinterpretation is to be further manipulated for whatever reason, including just discarding it, we need a way to divert the output of reevaluations. This is done through the creation of a BodyContent object and use of the setBodyContent() method, which is part of the BodyTag interface. BodyTag also provides the doInitBody() method which is invoked after setBodyContent() and before the first body evaluation provides an opportunity to interact with the body.

JSP.7.1.3.6 Cooperating Actions

Cooperating actions may offer the best way to describe a desired functionality. For example, one action may be used to describe information leading to the creation of a server-side object, while another action may use that object elsewhere in the page. These actions may cooperate explicitly, via scripting variables: one action creates an object and gives it a name; the other refers to the object through the name. Scripting variables are discussed briefly below.

Two actions can also cooperate implicitly. A flexible and convenient mechanism for action cooperation uses the nested structure of the actions to describe scoping. This is supported in the specification by providing each tag handler with its parent tag handler (if any) through the setParent() method. The findAncestorWithClass static method in TagSupport can then be used to locate a tag handler, and, once located, to perform valid operations on the tag handler.

JSP.7.1.3.7 Actions Defining Scripting Variables

A custom action may create server-side objects and make them available to scripting elements by creating or updating the scripting variables. The variables thus affected are part of the semantics of the custom action and are the responsability of the tag library author.

This information is used at JSP page translation time and can be described in one of two ways: directly in the TLD for simple cases, or through subclasses of TagExtraInfo. Either mechanism will indicate the names and types of the scripting variables.

At request time the tag handler will associate objects with the scripting variables through the pageContext object.

It is the responsibility of the JSP page translator to automatically supply the code required to do the “synchronization” between the pageObject values and the scripting variables.

JSP.7.2 Tag Libraries

A *tag library* is a collection of actions that encapsulate some functionality to be used from within a JSP page. A tag library is made available to a JSP page through a taglib directive that identifies the tag library via a URI (Universal Resource Identifier).

The URI identifying a tag library may be any valid URI as long as it can be used to uniquely identify the semantics of the tag library.

The URI identifying the tag library is associated with a *Tag Library Description* (TLD) file and with *tag handler* classes as indicated in Section JSP.7.3 below.

JSP.7.2.1 Packaged Tag Libraries

JSP page authoring tools and JSP containers are required to accept a tag library that is packaged as a JAR file. When deployed in a JSP container, the standard JAR conventions described in the Servlet 2.3 specification apply, including the conventions for dependencies on extensions.

Packaged tag libraries must have at least one tag library descriptor file. The JSP 1.1 specification allowed only a single TLD, in META-INF/taglib.tld, but in JSP 1.2 multiple tag libraries are allowed. See Section JSP.7.3.1 for how TLDs are identified.

JSP.7.2.2 Location of Java Classes

A tag library contains classes for instantiation at translation time and classes for instantiation at request time. The former include TagLibraryValidator and TagExtraInfo classes. The later include tag handler and event listener classes.

The usual conventions for Java classes apply: as part of a web application, they must reside either in a JAR file in the WEB-INF/lib directory, or in a directory in the WEB-INF/classes directory.

A JAR containing 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 JSP.7.3.7). The mapping between the URI and the TLD is explained further below.

JSP.7.2.3 Tag Library directive

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

A JSP container maps the URI used in the taglib directive into a Tag Library Descriptor in two steps: it resolves the URI into a TLD resource path, and then derives the TLD object from the TLD resource path.

If the JSP container cannot locate a TLD resource path for a given URI, a fatal translation error shall result. Similarly, it is a fatal translation error for a uri attribute value to resolve to two different TLD resource paths.

It is a fatal translation error for the taglib directive to appear after actions using the prefix introduced by it.

JSP.7.3 The Tag Library Descriptor

The Tag Library Descriptor (TLD) is an XML document that describes a tag library. The TLD for a tag library is used by a JSP container to interpret pages that include taglib directives referring to that tag library. The TLD is also used by JSP page authoring tools that will generate JSP pages that use a library, and by authors who do the same manually.

The TLD includes documentation on the library as a whole and on its individual tags, version information on the JSP container and on the tag library, and information on each of the actions defined in the tag library.

The TLD may name a TagLibraryValidator class that can validate that a JSP page conforms to a set of constraints expected by the tag library.

Each action in the library is described by giving its name, the class of its tag handler, information on any scripting variables created by the action, and information on attributes of the action. Scripting variable information can be given directly in the TLD or through a TagExtraInfo class. For each valid attribute there is an indication about whether it is mandatory, whether it can accept request-time expressions, and additional information.

A TLD file is useful for providing information on a tag library. It can be read by tools without instantiating objects or loader classes. Our approach conforms to the conventions used in other J2EE technologies.

The DTD for 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.

JSP.7.3.1 Identifying Tag Library Descriptors

Tag library descriptor files have names that use the extension “.tld”, and the extension indicates a tag library descriptor file. When deployed inside a JAR file, the tag library descriptor files must be in the META-INF directory, or a subdirectory of it. When deployed directly into a web application, the tag library descriptor files must always be in the WEB-INF directory, or some subdirectory of it.

The DTD for a TLD document is "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd"

JSP.7.3.2 TLD resource path

A URI in a taglib directive is mapped into a context relative path (as discussed in Section JSP.2.2.1). The context relative path is a URL without a protocol and host components that starts with “/” and is called the *TLD resource path*.

The TLD resource path is interpreted relative to the root of the web application and should resolve to a TLD file directly, or to a JAR file that has a TLD file at location META-INF/taglib.tld. If the TLD resource path is not one of these two cases, a fatal translation error will occur.

The URI describing a tag library is mapped to a TLD resource path though a *taglib map*, and a fallback interpretation that is to be used if the map does not contain the URI. The taglib map is built from an explicit taglib map in web.xml (described in Section JSP.7.3.3) that is extended with implicit entries deduced from packaged tag libraries in the web application (described in Section JSP.7.3.4), and implicit entries known to the JSP container. The fallback interpretation is targetted to a casual use of the mechanism, as in the development cycle of the Web Application; in that case the URI is interpreted as a direct path to the TLD (see Section JSP.7.3.6.2).

JSP.7.3.3 Taglib map in web.xml

The web.xml file can include an explicit taglib map between URIs and TLD resource paths described using the taglib elements of the Web Application Deployment descriptor in WEB-INF/web.xml, 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.
```

The body of the taglib-uri element may be either an absolute URI specification, or a relative URI as in Section JSP.2.2.1. There should be no entries in web.xml with the same taglib-uri value.

<*taglib-location*>

A taglib-location contains the location (as a resource) of the Tag Library Description File for the tag library.

```
<!ELEMENT taglib-location (#PCDATA) >

PCDATA ::= a resource location, as indicated in Section JSP.2.2.1, where to
find the Tag Library Descriptor file.
```

JSP.7.3.4 Implicit Map entries from TLDs

The taglib map described in web.xml is extended with new entries extracted from TLD files in the Web Application. The new entries are computed as follows:

- Each TLD file is examined. If it has a <uri> element, then a new <taglib> element is created, with a <taglib-uri> subelement whose value is that of the <uri> elemement, and with a <taglib-location> subelement that refers to the TLD file.
- If the created <taglib> element has a different <taglib-uri> to any in the taglib map, it is added.

This mechanism provides an automatic URI to TLD mapping as well as supporting multiple TLDs within a packaged JAR. Note that this functionality does not require explicitly naming the location of the TLD file, which would require a mechanism like the jar: protocol.

Note also that the mechanism does not add duplicated entries.

JSP.7.3.5 Implicit Map entries from the Container

The container may also add additional entries to the taglib map. As in the previous case, the entries are only added for URIs that are not present in the map. Conceptually the entries correspond to TLD describing these tag libraries.

These implicit map entries correspond to libraries that are known to the container, who is responsible for providing their implementation, either through tag handlers, or via the mechanism described in Section JSP.7.3.9.

JSP.7.3.6 Determining the TLD Resource Path

The TLD resource path can be determined from the uri attribute of a taglib directive as described below. In the explanation below an “absolute URI” is one that starts with a protocol and host, while a“relative URI specification” is as in section 2.5.2, i.e. one without the protocol and host part.

All steps are described as if they were taken, but an implementation can use a different implementation strategy as long as the result is preserved.

JSP.7.3.6.1 Computing TLD Locations

The taglib map generated in Sections JSP.7.3.3, JSP.7.3.4 and JSP.7.3.5 may contain one or more <taglib></taglib> entries. Each entry is identified by a TAGLIB_URI, which is the value of the <taglib-uri> subelement. This TAGLIB_URI may be an absolute URI, or a relative URI spec starting with “/” or one not starting with “/”. Each entry also defines a TAGLIB_LOCATION as follows:

- If the <taglib-location> subelement is some relative URI specification that starts with a “/” the TAGLIB_LOCATION is this URI.
- If the <taglib-location> subelement is some relative URI specification that does not start with “/”, the TAGLIB_LOCATION is the resolution of the URI relative to /WEB-INF/web.xml (the result of this resolution is a relative URI specification that starts with “/”).

JSP.7.3.6.2 Computing the TLD Resource Path

The following describes how to resolve a taglib directive to compute the TLD resource path. It is based on the value of the uri attribute of the taglib directive.

- If uri is ABS_URI, an absolute URI

Look in the taglib map for an entry whose TAGLIB_URI is ABS_URI. If found, the corresponding TAGLIB_LOCATION is the TLD resource path. If not found, a translation error is raised.

- If uri is ROOT_REL_URI, a relative URI that starts with "/"

Look in the taglib map for an entry whose TAGLIB_URI is ROOT_REL_URI. If found, the corresponding TAGLIB_LOCATION is the TLD resource path. If no such entry is found, ROOT_REL_URI is the TLD resource path.

- If uri is NOROOT_REL_URI, a relative URI that does not start with "/"

Look in the taglib map for an entry whose TAGLIB_URI is NOROOT_REL_URI. If found, the corresponding TAGLIB_LOCATION is the TLD resource path. If no such entry is found, resolve NOROOT_REL_URI relative to the current JSP page where the directive appears; that value (by definition, this is a relative URI specification that starts with "/") is the TLD resource path.

JSP.7.3.6.3 Usage Considerations

The explicit web.xml map provides a explicit description of the tag libraries that are being used in a web application.

The implicit map from TLDs means that a JAR file implementing a tag library can be dropped in and used immediatedly through its stable URIs.

The use of relative URI specifications in the taglib map enables very short names in the taglib directive. For example, if the map is:

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

then it can be used as:

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

Finally, the fallback rule allows a taglib directive to refer directly to the TLD. This arrangement is very convenient for quick development at the expense of less flexibility and accountability. For example, in the case above, it enables:

```
<%@ taglib uri="/WEB-INF/tlds/PRlibrary_1_4.tld" prefix="x" %>
```

JSP.7.3.7 Translation-Time Class Loader

The set of classes available at translation time is the same as that available at runtime: the classes in the underlying Java platform, those in the JSP container, and those in the class files in WEB-INF/classes, in the JAR files in WEB-INF/lib, and, indirectly those indicated through the use of the class-path attribute in the META-INF/MANIFEST file of these JAR files.

JSP.7.3.8 Assembling a Web Application

As part of the process of assembling a web application, the Application Assembler will create a WEB-INF/ directory, with appropriate lib/ and classes/ subdirectories, place JSP pages, Servlet classes, auxiliary classes, and tag libraries in the proper places, and create a WEB-INF/web.xml that ties everything together.

Tag libraries that have been delivered in the standard JAR format can be dropped directly into WEB-INF/lib. This automatically adds all the TLDs inside the JAR, making their URIs advertised in their <uri> elements visible to the URI to TLD map. The assembler may create taglib entries in web.xml for each of the libraries that are to be used.

Part of the assembly (and later the deployment) may create and/or change information that customizes a tag library; see Section JSP.7.6.3.

JSP.7.3.9 Well-Known URIs

A JSP container may “know of” some specific URIs and may provide alternate implementations for the tag libraries described by these URIs, but the user must see the behavior as that described by the required, portable tag library description described by the URI.

A JSP container must always use the mapping specified for a URI in the web.xml deployment descriptor if present. If the deployer wants to use the platform-specific implementation of the well-known URI, the mapping for that URI should be removed at deployment time.

JSP.7.4 The Tag Library Descriptor Format

This section describes the DTD for the JSP 1.2 version of the Tag Library Descriptor. The JSP 1.2 version has information added from the JSP 1.1 version, as well as a few changes to element names made to improve consistency with other specifications.

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

Notation

```
<!NOTATION WEB-JSPTAGLIB.1_2 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/JSP/TagLibraryDescriptor"
>
```

A taglib element also has several subelements that define:

tlib-version	the version of the tag library implementation
jsp-version	the mandatory version of JSP specification the tag library depends upon
short-name	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 small-icon that can be used by tools.
large-icon	Optional large-icon that can be used by tools.
description	a string describing the “use” of this taglib.
validator	Optional TagLibraryValidator information.
listener	Optional event listener specification

```
<!ELEMENT taglib
    (tlib-version, jsp-version,
     short-name, uri?, display-name?, small-icon?, large-icon?
     description?, validator?, listener*, tag+) >
```

<tlib-version>

Describes the version (number) of the tag library.
The syntax is:

```
<!ELEMENT tlib-version (#PCDATA) >
#PCDATA ::= [0-9]*{ "."[0-9] }0..3
```

<jsp-version>

Describes the JSP specification version (number) this tag library requires in order to function. This element is mandatory
The syntax is:

```
<!ELEMENT jsp-version  (#PCDATA) >
#PCDATA ::= [0-9]*{ "."[0-9] }0..3.
```

<short-name>

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 short-name       (#PCDATA) >
#PCDATA ::= NMTOKEN
```

<uri>

Defines a public URI that uniquely identifies this version of the tag library.

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

<description>

Defines an arbitrary text string describing the tag library, variable, attribute or validator.

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

<validator>

Defines an optional TagLibraryValidator that can be used to validate the conformance of a JSP page to using this tag library. A validator may have some optional initialization parameters.
The **validator** may have several subelements defining:

validator-class	the class implementing javax.servlet.jsp.tagext.TagLibraryValidator
init-param	optional initialization parameters
description	an optional description of the validator.

The element syntax is as follows:

<!ELEMENT **validator** (validator-class, init-param*, description?) >

<validator-class>
Defines the class of the optional TagLibraryValidator.

<!ELEMENT **validator-class** (#PCDATA) >

<init-param>
Defines an initialization parameter.
The **init-param** may have several subelements defining:

param-name	the name of the parameter
param-value	the value of the parameter
description	optional description of the parameter

The element syntax is as follows:

<!ELEMENT **init-param** (param-value, param-value, description?) >

<param-name>
The name of a parameter.

<!ELEMENT **param-name** (#PCDATA) >

<param-value>
The value of a parameter.

<!ELEMENT **param-value** (#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.
The common way to describe the semantics of a specific custom action that are observable by other custom actions is the implementation class of the tag handler in the tag-class element. But the description element can also be used to indicate a type that further constraints those operations. The type can be either void or a subtype of the tag handler implementation class. This information can be used by a specialized container for a specific well known tag libraries; see Section JSP.7.3.9.
The tag element has one attribute:

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

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

name	the unique action name
tag-class	the tag handler class implementing javax.servlet.jsp.tagext.Tag
tei-class	an optional subclass of javax.servlet.jsp.tagext.TagExtraInfo
body-content	the body content type
display-name	A short name that is intended to be displayed by tools.
small-icon	Optional large-icon that can be used by tools.
large-icon	Optional large-icon that can be used by tools.
description	Optional tag-specific information.
variable	Optional scripting variable information.
attribute	all attributes of this action
example	optional example of the use of this tag.

The element syntax is as follows:

```
<!ELEMENT tag
    (name, tag-class, tei-class?,
    body-content?, display-name?, small-icon?, large-icon?,
    description?, variable*, attribute*, example?) >
```

<tag-class>

Defines the tag handler implementation class for this custom action. The class must implement the javax.serlvet.jsp.tagext.Tag interface. This element is required.
The syntax is:

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

#PCDATA ::= fully qualified Java class name.
```

<tei-class>

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

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

<body-content>

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. No quoting is performed.
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 body-content (#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 a path within the tag library relative to the location of the TLD. 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 a path within the tag library relative to the location of the TLD. 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.
variable-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.

description	optional description of the variable.

The syntax is:

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

<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) >
```

<variable-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)
description	optional description of the attribute

The syntax is:

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

<*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 attribute’s value. For literal values (rtexprvalue is false) the type is always java.lang.String.
If the rtexprvalue element is true, then the type defines the return type expected from any scriptlet expression specified as the value of this attribute.
The value of this attribute should match that of the underlying JavaBean component property.
The syntax is:

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

An example is:

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

<example>

The content of this element is intended to be an example of how to use the tag. This element is not intepreted by the JSP container and has no effect on the semantics of the tag.

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

JSP.7.5 Validation

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

- Request-time semantics; e.g. a subelement may require the information from some enclosing element at request-time .
- Authoring-tool support; e.g. a tool may require an ordering in the actions.
- Methodological constraints; e.g. a development group may want to constrain the way some features are used.

Validation can be done either at translation-time or at request-time. In general translation-time validation provides a better user experience, and the JSP 1.2 specification provides a very flexible translation-time validation mechanism.

JSP.7.5.1 Translation-Time Mechanisms

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

JSP.7.5.1.1 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 element can be used to indicate that an action must be empty.

All constraints described in the TLD must be enforced. A tag library author can assume that the tag handler instance corresponds to an action that satisfies all constraints indicated in the TLD.

JSP.7.5.1.2 Validator Classes

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

The JSP container may optionally uniquely identify all XML elements in the XML view of a JSP page through a jsp:id attribute. This attribute can be used to provide better information on the location of an error.

The validator class mechanism is new to the JSP 1.2 specification. A TagLibraryValidator can be passed some initialization parameters in the TLD. This eases the reuse of validator classes. We expect that validator classes will be written based on different XML schema mechanisms (DTDs, XSchema, Relaxx, others). Standard validator classes may be incorporated into a later version of the JSP specification if a clear schema standard appears at some point.

JSP.7.5.1.3 TagExtraInfo Class Validation

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 should use the validator class mechanism.

JSP.7.5.2 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.

JSP.7.6 Conventions and Other Issues

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

JSP.7.6.1 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 to define the desired objects.

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 an implementation dependency in making these objects available. For example, they may be providing access to some functionality that exists only in a particular implementation. This can be done by having the tag extension class test at run-time for the existence of some implementation dependent feature and raise a run-time error (this, of course, makes the page not J2EE compliant).

This mechanism, together with the access to metadata information allows for vendors to innovate within the standard.

Note: if a feature is added to a JSP specification, and a vendor also provides that feature through its vendor-specific mechanism, the standard mechanism, as indicated in the JSP specification will “win”. This means that vendor-specific mechanisms can slowly migrate into the specification as they prove their usefulness.

JSP.7.6.2 Access to Vendor-Specific information

If a vendor wants to associate some information that is not described in the current version of the TLD with some tag library, it can do so by inserting the information in a document it controls, inserting the document in the WEB-INF portion of the 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.

JSP.7.6.3 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 JSP.8

JSP Container

This chapter describes the contracts between a JSP container and a JSP page. The precompilation protocol (see Section JSP.8.4) is also presented here.

The information in this chapter is independent of the Scripting Language used in the JSP page. Chapter JSP.6 describes information specific to when the language attribute of the page directive has “java” as its value.).

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

JSP.8.1 JSP Page Model

A JSP page is represented at execution time by a JSP page implementation object and is executed by a JSP container. The JSP page implementation object is a servlet. The JSP container delivers requests from a client to a JSP page implementation object and responses from the JSP page implementation object to the client.

The JSP page describes how to create a *response* object from a *request* object for a given protocol, possibly creating and/or using some other objects in the process . A JSP page may also indicate how some events are to be handled. In JSP 1.2 only *init* and *destroy* events are allowed events.

JSP.8.1.1 Protocol Seen by the Web Server

The JSP container locates the appropriate instance of the JSP page implementation class and delivers requests to it using the Servlet protocol. A JSP container may need to create such a class dynamically from the JSP page source before delivering request and response objects to it.

The Servlet class defines the contract between the JSP container and the JSP page implementation class. When the HTTP protocol is used, the contract is described by the HttpServlet class. Most JSP pages use the HTTP protocol, but other protocols are allowed by this specification.

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

JSP.8.1.1.1 Protocol Seen by the JSP Page Author

The JSP specification defines the contract between the JSP container and the JSP page author. This contract defines the assumptions an author can make for the actions described in the JSP page.

The main portion of this contract is the *_jspService()* method that is generated automatically by the JSP container from the JSP page. The details of this contract are provided in Chapter JSP.6.

The contract also describes how a JSP author can indicate what actions will be taken when the *init()* and *destroy()* methods of the page implementation occur. In JSP 1.2 this is done by defining methods with names *jspInit()* and *jspDestroy()* in a declaration scripting element in the JSP page. The *jspInit()* method, if present, will be called to prepare the page before the first request is delivered. Similarly a JSP container can reclaim resources used by a JSP page when a request is not being serviced by the JSP page by invoking its *jspDestroy()* method, if present.

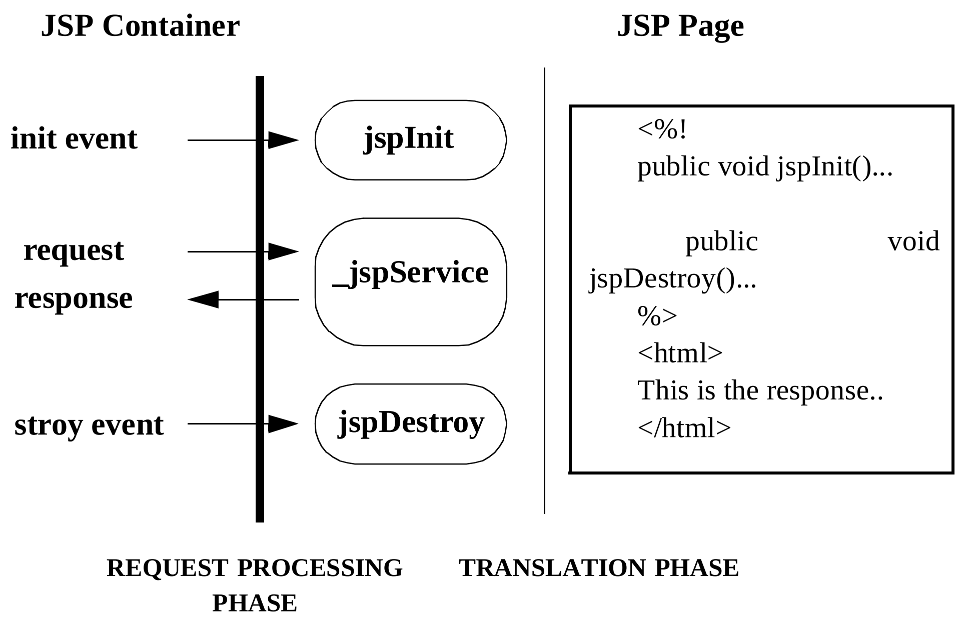
A JSP page author may **not** (re)define Servlet methods through a declaration scripting element.

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

JSP.8.1.1.2 The HttpJspPage Interface

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

Figure J2EE.8.1 Contracts between a JSP Page and a JSP Container.



The involved contracts are shown in Figure J2EE.8.1. We now revisit this whole process in more detail.

JSP.8.2 JSP Page Implementation Class

The JSP container creates a JSP page implementation class for each JSP page.

The name of the JSP page implementation class is implementation dependent.

The JSP Page implementation object belongs to an implementation-dependent named package. The package used may vary between one JSP and another, so minimal assumptions should be made. The unnamed package should not be used without an explicit “import” of the class.

The JSP container may create the implementation class for a JSP page, or a superclass may be provided by the JSP page author through the use of the extends attribute in the page directive.

The extends mechanism is available for sophisticated users. It should be used with extreme care as it restricts decisions that a JSP container can make. It may restrict efforts to emprove performance, for example.

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 support classes. If the JSP page implementation class is packaged into a WAR, any dependant classes will have to be included so it will be portable across all JSP containers.

A JSP page author writes a JSP page expecting that the client and the server will communicate using a certain protocol. The JSP container must guarantee that requests 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.

JSP.8.2.1 API Contracts

The contract between the JSP container and a Java class implementing a JSP page corresponds to the Servlet interface. Refer to the Servlet 2.3 specification for details.

The responsibility for adhering to this contract rests on the JSP container implementation if the JSP page does not use the *extends* attribute of the *jsp* directive. If the *extends* attribute of the *jsp* directive is used, the JSP page author must guarantee that the superclass given in the extends attribute supports this contract.

Table JSP.8-1 *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()

Table JSP.8-1 *How the JSP Container Processes JSP Pages*

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

JSP.8.2.2 Request and Response Parameters

As shown in Table JSP.8-1, 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.Servlet-Response. The interface must define a protocol-dependent response contract between the JSP container and the class that implements the JSP page.

The request and response interfaces together describe a protocol-dependent contract between the JSP container and the class that implements the JSP page. The HTTP contract is defined by the javax.servlet.http.HttpServletRequest and javax.servlet.http.HttpServletResponse interfaces.

The JspPage interface refers to these methods, but cannot describe syntactically the methods involving the Servlet(Request,Response) subtypes. However, interfaces for specific protocols that extend JspPage can, just as HttpJspPage describes them for the HTTP protocol.

JSP containers that conform to this specification (in both JSP page implementation classes and JSP container runtime) must implement the request and response interfaces for the HTTP protocol as described in this section.

JSP.8.2.3 Omitting the `extends` Attribute

If the extends attribute of the page directive (seeSection 2.10.1) in a JSP page is not used, the JSP container can generate any class that satisfies the contract described in Table JSP.8-1, when it transforms the JSP page.

In the following code examples, Code Example 8.1 illustrates a generic HTTP superclass named ExampleHttpSuper. Code Example 8.2 shows a subclass named _jsp1344 that extends ExampleHttpSuper and is the class generated from the JSP page. By using separate _jsp1344 and ExampleHttpSuper classes, the JSP page translator does not need to discover whether the JSP page includes a declaration with jspInit() or jspDestroy(). This significantly simplifies the implementation.

Code Example 8.1 A Generic HTTP Superclass

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

/**
 * An example of a superclass for an HTTP JSP class
 */

abstract class ExampleHttpSuper implements HttpJspPage {
    private ServletConfig config;

final public void init(ServletConfig config) throws ServletException {
    this.config = config;
    jspInit();

public void jspInit() {
}

public void jspDestroy() {
}

}

final public ServletConfig getServletConfig() {
    return config;
}

// This one is not final so it can be overridden by a more precise method
public String getServletInfo() {
    return “A Superclass for an HTTP JSP”; // maybe better?
}
```

```
final public void destroy() {
    jspDestroy();
}

/**
* The entry point into service.
*/

final public void service(ServletRequest req, ServletResponse res)
    throws ServletException, IOException {

    // casting exceptions will be raised if an internal error.
    HttpServletRequest request = (HttpServletRequest) req;
    HttpServletResponse response = (HttpServletResponse) res;

    _jspService(request, 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 8.2 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.

        }

    }
```

JSP.8.2.4 Using the extends Attribute

If the JSP page author uses extends, the generated class is identical to the one shown in Code Example 8.2, 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.

JSP.8.3 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 once 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 _jspService method as in the following example:

```
import javax.servlet.jsp.JspWriter;

static JspFactory _jspFactory = JspFactory.getDefaultFactory();

_jspService(<SRequest> request, <SResponse> response) {

    // initialization of implicit variables...
    PageContext pageContext = _jspFactory.createPageContext(
                        this,
                        request,
                        response,
                        false,
                        PageContext.DEFAULT_BUFFER,
                        false
                    );
    JSPWriter out = pageContext.getOut();
    // ....
    // .... the body goes here using "out"
    // ....
    out.flush();
}
```

You can find the complete listing of javax.servlet.jsp.JspWriter in Chapter JSP.9.

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

JSP.8.4 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 is related but not the same as the notion of compiling a JSP page into a Servlet class (Appendix JSP.A).

JSP.8.4.1 Request Parameter Names

All request parameter names that start with the prefix "jsp" are reserved by the JSP specification and should not be used by any user or implementation except as indicated by the specification.

All JSPs pages should ignore (not depend on) any parameter that starts with "jsp_"

JSP.8.4.2 Precompilation Protocol

A request to a JSP page that has a request parameter with name "jsp_precompile" is a *precompilation request*. The "jsp_precompile" parameter may have no value, or may have values "true" or "false". In all cases, the request should not be delivered to the JSP page.

The intention of the precompilation request is that of a suggestion to the JSP container to *precompile* the JSP page into its JSP page implementation class. The suggestion is conveyed by giving the parameter the value "true" or no value, but note that the request can be ignored.

For example:

1. ?jsp_precompile
2. ?jsp_precompile="true"
3. ?jsp_precompile="false"
4. ?foobar="foobaz"&jsp_precompile="true"
5. ?foobar="foobaz"&jsp_precompile="false"

   1, 2 and 4 are legal; the request will not be delivered to the page. 3 and 5 are legal; the request will not be delivered to the page.
6. ?jsp_precompile="foo"

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

CHAPTER JSP.9
Core API

This chapter describes the javax.servlet.jsp package. The chapter includes content that is generated automatically from javadoc embedded into the actual Java classes and interfaces. This allows the creation of a single, authoritative, specification document.

The javax.servlet.jsp package contains a number of classes and interfaces that describe and define the contracts between a JSP page implementation class and the runtime environment provided for an instance of such a class by a conforming JSP container.

JSP.9.1 JSP Page Implementation Object Contract

This section describes the basic contract between a JSP Page implementation object and its container. The main contract is defined by the classes JspPage and HttpJspPage. The JspFactory class describes the mechanism to portably instantiate all needed runtime objects, and JspEngineInfo provides basic information on the current JSP container.

None of the classes described here are intended to be used by JSP page authors; an example of how these classes may be used is included elsewhere in this chapter.

JSP.9.1.1 JspPage

Syntax

public interface JspPage extends javax.servlet.Servlet

All Known Subinterfaces: HttpJspPage

All Superinterfaces: javax.servlet.Servlet

Description

The JspPage interface describes the generic interaction that a JSP Page Implementation class must satisfy; pages that use the HTTP protocol are described by the HttpJspPage interface.

Two plus One Methods

The interface defines a protocol with 3 methods; only two of them: jspInit() and jspDestroy() are part of this interface as the signature of the third method: _jspService() depends on the specific protocol used and cannot be expressed in a generic way in Java.

A class implementing this interface is responsible for invoking the above methods at the appropriate time based on the corresponding Servlet-based method invocations.

The jspInit() and jspDestroy() methods can be defined by a JSP author, but the _jspService() method is defined automatically by the JSP processor based on the contents of the JSP page.

_jspService()

The _jspService()method corresponds to the body of the JSP page. This method is defined automatically by the JSP container and should never be defined by the JSP page author.

If a superclass is specified using the extends attribute, that superclass may choose to perform some actions in its service() method before or after calling the _jspService() method. See using the extends attribute in the JSP_Engine chapter of the JSP specification.

The specific signature depends on the protocol supported by the JSP page.

```
 public void _jspService(ServletRequestSubtype request,
                         ServletResponseSubtype response)
        throws ServletException, IOException;
```

JSP.9.1.1.1 Methods

public void **jspDestroy**()

The jspDestroy() method is invoked when the JSP page is about to be destroyed. A JSP page can override this method by including a definition for it in a declaration element. A JSP page should redefine the destroy() method from Servlet.

public void **jspInit**()

The jspInit() method is invoked when the JSP page is initialized. It is the responsibility of the JSP implementation (and of the class mentioned by the extends attribute, if present) that at this point invocations to the getServlet-Config() method will return the desired value. A JSP page can override this method by including a definition for it in a declaration element. A JSP page should redefine the init() method from Servlet.

JSP.9.1.2 HttpJspPage

Syntax

public interface HttpJspPage extends JspPage

All Superinterfaces: JspPage, javax.servlet.Servlet

Description

The HttpJspPage interface describes the interaction that a JSP Page Implementation Class must satisfy when using the HTTP protocol.

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

See Also: JspPage

JSP.9.1.2.2 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

JSP.9.1.3 JspFactory

Syntax

public abstract class JspFactory

Description

The JspFactory is an abstract class that defines a number of factory methods available to a JSP page at runtime for the purposes of creating instances of various interfaces and classes used to support the JSP implementation.

A conformant JSP Engine implementation will, during it's initialization instantiate an implementation dependent subclass of this class, and make it globally available for use by JSP implementation classes by registering the instance created with this class via the static setDefaultFactory() method.

The PageContext and the JspEngineInfo classes are the only implementation-dependent classes that can be created from the factory.

JspFactory objects should not be used by JSP page authors.

JSP.9.1.3.3 Constructors

public **JspFactory**()

JSP.9.1.3.4 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() method of a JSP implementation class in order to obtain a PageContext object for the request being processed.

Invoking this method shall result in the PageContext.initialize() method being invoked. The PageContext returned is properly initialized.

All PageContext objects obtained via this method shall be released by invoking releasePageContext().

Parameters:
servlet - the requesting servlet

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

JSP.9.1.4 JspEngineInfo

Syntax
public abstract class JspEngineInfo

Description

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

JSP.9.1.4.5 Constructors

public **JspEngineInfo**()

JSP.9.1.4.6 Methods

public abstract java.lang.String **getSpecificationVersion**()

Return the version number of the JSP specification that is supported by this JSP engine.

Specification version numbers that consists of positive decimal integers separated by periods ".", for example, "2.0" or "1.2.3.4.5.6.7". This allows an extensible number to be used to represent major, minor, micro, etc versions. The version number must begin with a number.

Returns: the specification version, null is returned if it is not known

JSP.9.2 Implicit Objects

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

JSP.9.2.1 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. Implicit objects are added the pageContext automatically.

The PageContext class is an abstract class, designed to be extended to provide implementation dependent implementations thereof, by conformant JSP engine runtime environments. A PageContext instance is obtained by a JSP 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 objects representing scopes. 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().

JSP.9.2.1.7 Fields

public static final java.lang.String **APPLICATION**

Name used to store ServletContext in PageContext name table.

public static final int **APPLICATION_SCOPE**

Application scope: named reference remains available in the ServletContext until it is reclaimed.

public static final java.lang.String **CONFIG**

Name used to store ServletConfig in PageContext name table.

public static final java.lang.String **EXCEPTION**

Name used to store uncaught exception in ServletRequest attribute list and PageContext name table.

public static final java.lang.String **OUT**

Name used to store current JspWriter in PageContext name table.

public static final java.lang.String **PAGE**

Name used to store the Servlet in this PageContext's nametables.

public static final int **PAGE_SCOPE**

Page scope: (this is the default) the named reference remains available in this PageContext until the return from the current Servlet.service() invocation.

public static final java.lang.String **PAGECONTEXT**

Name used to store this PageContext in it's own name table.

public static final java.lang.String **REQUEST**

Name used to store ServletRequest in PageContext name table.

public static final int **REQUEST_SCOPE**

Request scope: the named reference remains available from the Servlet-Request associated with the Servlet until the current request is completed.

public static final java.lang.String **RESPONSE**

Name used to store ServletResponse in PageContext name table.

public static final java.lang.String **SESSION**

Name used to store HttpSession in PageContext name table.

public static final int **SESSION_SCOPE**

Session scope (only valid if this page participates in a session): the named reference remains available from the HttpSession (if any) associated with the Servlet until the HttpSession is invalidated.

JSP.9.2.1.8 Constructors

public **PageContext**()

JSP.9.2.1.9 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 _jsp-Service(...) 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 specified scope or null if not found.

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

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

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.

This method is kept for backwards compatiblity reasons. Newly generated code should use PageContext.handlePageException(Throwable).

Parameters:
e - the exception to be handled

Throws:
ServletException, IOException

NullPointerException - if the exception is null

SecurityException - if target resource cannot be accessed by caller

See Also: public abstract void handlePageException(java.lang.Throwable t)

public abstract void **handlePageException**(java.lang.Throwable t)

This method is identical to the handlePageException(Exception), except that it accepts a Throwable. This is the preferred method to use as it allows proper implementation of the errorpage semantics.

This method is intended to process an unhandled "page" level exception by redirecting the exception to either the specified error page for this JSP, or if none was specified, to perform some implementation dependent action.

A JSP implementation class shall typically clean up any local state prior to invoking this and will return immediately thereafter. It is illegal to generate any output to the client, or to modify any ServletResponse state after invoking this call.

Parameters:
t - the throwable to be handled

Throws:
ServletException, IOException

NullPointerException - if the exception is null

SecurityException - if target resource cannot be accessed by caller

See Also: public abstract void handlePageException(java.lang.Exception e)

public abstract void **include**(java.lang.String relativeUrlPath)

Causes the resource specified to be processed as part of the current ServletRequest and ServletResponse being processed by the calling Thread. The output of the target resources processing of the request is written directly to the ServletResponse output stream.

The current JspWriter "out" for this JSP is flushed as a side-effect of this call, prior to processing the include.

If the *relativeUrlPath* begins with a "/" then the URL specified is calculated relative to the DOCROOT of the ServletContext for this JSP. If the path does not begin with a "/" then the URL specified is calculated relative to the URL of the request that was mapped to the calling JSP.

It is only valid to call this method from a Thread executing within a _jspService(...) method of a JSP.

Parameters:
relativeUrlPath - specifies the relative URL path to the target resource to be included

Throws:
ServletException, 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 method is called to initialize an uninitialized PageContext so that it may be used by a JSP Implementation class to service an incoming request and response within it's _jspService() method.

This method is typically called from JspFactory.getPageContext() in order to initialize state.

This method is required to create an initial JspWriter, and associate the “out” name in page scope with this newly created object.

This method should not be used by page or tag library authors.

Parameters:
servlet - The Servlet that is associated with this PageContext

request - The currently pending request for this Servlet

response - The currently pending response for this Servlet

errorPageURL - The value of the errorpage attribute from the page directive or null

needsSession - The value of the session attribute from the page directive

bufferSize - The value of the buffer attribute from the page directive

autoFlush - The value of the autoflush attribute from the page directive

Throws:
IOException - during creation of JspWriter

IllegalStateException - if out not correctly initialized

IllegalArgumentException

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 Jsp-Factory.releasePageContext().

Subclasses shall envelope this method.

This method should not be used by page or tag library 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

JSP.9.2.2 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 is not buffered. If the page is not buffered, output written to this JspWriter object will be written through to the PrintWriter directly, which will be created if necessary by invoking the getWriter() method on the response object. But if the page is buffered, the PrintWriter object will not be created until the buffer is flushed and operations like setContentType() are legal. Since this flexibility simplifies programming substantially, buffering is the default for JSP pages.

Buffering raises the issue of what to do when the buffer is exceeded. Two approaches can be taken:

- •Exceeding the buffer is not a fatal error; when the buffer is exceeded, just flush the output.
- •Exceeding the buffer is a fatal error; when the buffer is exceeded, raise an exception.

Both approaches are valid, and thus both are supported in the JSP technology. The behavior of a page is controlled by the autoFlush attribute, which defaults to true. In general, JSP pages that need to be sure that correct and complete data has been sent to their client may want to set autoFlush to false, with a typical case being that where the client is an application itself. On the other hand, JSP pages that send data that is meaningful even when partially constructed may want to set autoFlush to true; such as when the data is sent for immediate display through a browser. Each application will need to consider their specific needs.

An alternative considered was to make the buffer size unbounded; but, this had the disadvantage that runaway computations would consume an unbounded amount of resources.

The “out” implicit variable of a JSP implementation class is of this type. If the page directive selects autoflush=“true” then all the I/O operations on this class shall automatically flush the contents of the buffer if an overflow condition would result if the current operation were performed without a flush. If autoflush="false" then all the I/O operations on this class shall throw an IOException if performing the current operation would result in a buffer overflow condition.

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

JSP.9.2.2.10 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

JSP.9.2.2.11 Constructors

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

protected constructor.

JSP.9.2.2.12 Methods

public abstract void **clear**()

Clear the contents of the buffer. If the buffer has been already been flushed then the clear operation shall throw an IOException to signal the fact that some data has already been irrevocably written to the client response stream.

Throws:
IOException **-** If an I/O error occurs

public abstract void **clearBuffer**()

Clears the current contents of the buffer. Unlike clear(), this method will not throw an IOException if the buffer has already been flushed. It merely clears the current content of the buffer and returns.

Throws:
IOException **-** If an I/O error occurs

public abstract void **close**()

Close the stream, flushing it first

This method needs not be invoked explicitly for the initial JspWriter as the code generated by the JSP container will automatically include a call to close().

Closing a previously-closed stream, unlike flush(), has no effect.

Overrides: java.io.Writer.close() in class java.io.Writer

Throws:
IOException - If an I/O error occurs

public abstract void **flush**()

Flush the stream. If the stream has saved any characters from the various write() methods in a buffer, write them immediately to their intended destination. Then, if that destination is another character or byte stream, flush it. Thus one flush() invocation will flush all the buffers in a chain of Writers and OutputStreams.

The method may be invoked indirectly if the buffer size is exceeded.

Once a stream has been closed, further write() or flush() invocations will cause an IOException to be thrown.

Overrides: java.io.Writer.flush() in class java.io.Writer

Throws:
IOException - If an I/O error occurs

public int **getBufferSize**()

This method returns the size of the buffer used by the JspWriter.

Returns: the size of the buffer in bytes, or 0 is unbuffered.

public abstract int **getRemaining**()

This method returns the number of unused bytes in the buffer.

Returns: the number of bytes unused in the buffer

public boolean **isAutoFlush**()

This method indicates whether the JspWriter is autoFlushing.

Returns: if this JspWriter is auto flushing or throwing IOExceptions on buffer overflow conditions

public abstract void **newLine**()

Write a line separator. The line separator string is defined by the system property line.separator, and is not necessarily a single newline (’\n’) character.

Throws:

IOException - If an I/O error occurs

public abstract void **print**(boolean b)

Print a boolean value. The string produced by java.lang.String.valueOf(boolean) is 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

JSP.9.3 An Implementation Example

An instance of an implementation dependent subclass of this abstract base class can be created by a JSP implementation class at the beginning of it's `_jspService()` method via an implementation default `JspFactory`.

Here is one example of how to use these classes

```
 public class foo implements Servlet {
 // ...
public void _jspService(HttpServletRequest request,
HttpServletResponse response)
     throws IOException, ServletException {
   JspFactory  factory     = JspFactory.getDefaultFactory();
   PageContext pageContext = factory.getPageContext(
this,
request,
response,
null,  // errorPageURL
false, // needsSession
JspWriter.DEFAULT_BUFFER,
true   // autoFlush
      );
   // initialize implicit variables for scripting env ...
   HttpSession session = pageContext.getSession();
   JspWriter   out     = pageContext.getOut();
   Object      page    = this;
   try {
      // body of translated JSP here ...
   } catch (Exception e) {
      out.clear();
      pageContext.handlePageException(e);
   } finally {
      out.close();
  factory.releasePageContext(pageContext);
   }
}
```

JSP.9.4 Exceptions

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

JSP.9.4.1 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.

JSP.9.4.1.13 Constructors

public **JspException**()

Construct a JspException

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

Constructs a new JSP exception with the specified message. The message can be written to the server log and/or displayed for the user.

Parameters:
msg - a String specifying the text of the exception message

public **JspException**(java.lang.String message, java.lang.Throwable rootCause)

Constructs a new JSP exception when the JSP needs to throw an exception and include a message about the “root cause” exception that interfered with its normal operation, including a description message.

Parameters:
message - a String containing the text of the exception message

rootCause - the Throwable exception that interfered with the servlet’s normal operation, making this servlet exception necessary

public **JspException**(java.lang.Throwable rootCause)

Constructs a new JSP exception when the JSP needs to throw an exception and include a message about the “root cause” exception that interfered with its normal operation. The exception’s message is based on the localized message of the underlying exception.

This method calls the getLocalizedMessage method on the Throwable exception to get a localized exception message. When subclassing JspException, this method can be overridden to create an exception message designed for a specific locale.

Parameters:
rootCause - the Throwable exception that interfered with the JSP’s normal operation, making the JSP exception necessary

JSP.9.4.1.14 Methods

public java.lang.Throwable **getRootCause**()

Returns the exception that caused this JSP exception.

Returns: the Throwable that caused this JSP exception

JSP.9.4.2 JspTagException

Syntax

public class JspTagException extends JspException

All Implemented Interfaces: java.io.Serializable

Description

Exception to be used by a Tag Handler to indicate some unrecoverable error. This error is to be caught by the top level of the JSP page and will result in an error page.

JSP.9.4.2.15 Constructors

public **JspTagException**()

No message

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

Constructor with a message.

CHAPTER JSP.10

Tag Extension API

This chapter describes the details of tag handlers and other tag extension classes as well as methods that are available to access the Tag Library Descriptor files. This complements a previous chapter that described the Tag Library Descriptor files formats and their use in taglib directives.

This chapter includes content that is generated automatically from javadoc embedded into the actual Java classes and interfaces. This allows the creation of a single, authoritative, specification document.

Custom actions can be used by JSP authors and authoring tools to simplify writing JSP pages. A custom action can be either an empty or a non-empty action.

An empty tag has no body. There are two equivalent syntaxes, one with separate start and an end tag, and one where the start and end tags are combined. The two following examples are identical:

```
<x:foo att=“myObject” />
<x:foo att=“myObject” ></foo>
```

A non-empty tag has a start tag, a body, and an end tag. A prototypical example is of the form:

```
<x:foo att=“myObject” >
  BODY
</x:foo/>
```

The JavaServer Pages(tm) (JSP) 1.2 specification provides a portable mechanism for the description of tag libraries containing:

- •A Tag Library Descriptor (TLD)
- •A number of Tag handler classes defining request-time behavior
- •A number of classes defining translation-time behavior
- •Additional resources used by the classes

This chapter is organized in three sections. The first section presents the basic tag handler classes. The second section describes the more complex tag handlers that need to access their body evaluation. The last section looks at translation-time issues.

JSP.10.1 Simple Tag Handlers

This section introduces the notion of a tag handler and describes the simplest types of tag handler.

Tag Handler

A tag handler is a run-time, container-managed, object that evaluates custom actions during the execution of a JSP page. A tag handler supports a protocol that allows the JSP container to provide good integration of the server-side actions within a JSP page.

A tag handler is created initially using a zero argument constructor on its corresponding class; the method java.beans.Beans.instantiate() is not used.

A tag handler has some properties that are exposed to the page as attributes on an action; these properties are managed by the JSP container (via generated code). The setter methods used to set the properties are discovered using the JavaBeans introspector machinery.

The protocol supported by a tag handler provides for passing of parameters, the evaluation and reevaluation of the body of the action, and for getting access to objects and other tag handlers in the JSP page.

A tag handler instance is responsible for processing one request at a time. It is the responsability of the JSP container to enforce this.

Additional translation time information associated with the action indicates the name of any scripting variables it may introduce, their types and their scope. At specific moments, the JSP container will automatically synchronize the Page-Context information with variables in the scripting language so they can be made available directly through the scripting elements.

Properties

A tag handler has some properties. All tag handlers have a *pageContext* property for the JSP page where the tag is located, and a *parent* property for the tag handler to the closest enclosing action. Specific tag handler classes may have additional properties.

All attributes of a custom action must be JavaBeans component properties, although some properties may not be exposed as attributes. The attributes that are visible to the JSP translator are exactly those listed in the Tag Library Descriptor (TLD).

All properties of a tag handler instance exposed as attributes will be initialized by the container using the appropriate setter methods before the instance can be used to perform the action methods. It is the responsibility of the JSP container to invoke the appropriate setter methods to initialize these properties. It is the responsability of user code, be it scriptlets, JavaBeans code, or code inside custom tags, to not invoke these setter methods, as doing otherwise would interfere with the container knowledge.

The setter methods that should be used when assigning a value to an attribute of a custom action are determined by using the JavaBeans introspector on the tag handler class, then use the setter method associated with the property that has the same name as the attribute in question. An implication (unclear in the JavaBeans specification) is that there is only one setter per property.

Unspecified attributes/properties should not be set (using a setter method).

Once properly set, all properties are expected to be persistent, so that if the JSP container ascertains that a property has already been set on a given tag handler instance, it needs not set it again. User code can access property information and access and modify tag handler internal state starting with the first action method (doStartTag) up until the last action method (doEndTag or doFinally for tag handlers implementing TryCatchFinally).

Tag Handler as a Container-Managed Object

Since a tag handler is a container managed object, the container needs to maintain its references; specifically, user code should not keep references to a tag handler except between the start of the first action method (doStartTag()) and the end of the last action method (doEndTag() or doFinally() for those tags that implement TryCatchFinally).

The restrictions on references to tag handler objects and on modifying attribute properties gives the JSP container substantial freedom in effectively managing tag handler objects to achieve different goals. For example, a container may implementing different pooling strategies to minimize creation cost, or may hoist setting of properties to reduce cost when a tag handler is inside another iterative tag.

Conversions

A tag handler implements an action; the JSP container must follow the type conversions described in Section 2.13.2 when assigning values to the attributes of an action.

Empty and Non-Empty Actions

An empty action has no body; it may use one of two syntaxes: either <foo/> or <foo></foo>. Since empty actions have no body the methods related to body manipulation are not invoked. There is a mechanism in the Tag Library Descriptor to indicate that a tag can only be used to write empty actions; when used, non-empty actions using that tag will produce a translation error.

A non-empty action has a body.

The Tag Interface

A Tag handler that does not want to process its body can implement just the Tag interface. A tag handler may not want to process its body because it is an empty tag or because the body is just to be “passed through”.

The Tag interface includes methods to provide page context information to the Tag Handler instance, methods to handle the life-cycle of tag handlers, and two main methods for performing actions on a tag: doStartTag() and doEndTag(). The method doStartTag() is invoked when encountering the start tag and its return value indicates whether the body (if there is any) should be skipped, or evaluated and passed through to the current response stream. The method doEndTag() is invoked when encountering the end tag; its return value indicates whether the rest of the page should continue to be evaluated or not.

If an exception is encountered during the evaluation of the body of a tag, its doEndTag method will not be evaluated. See the TryCatchFinally tag for methods that are guaranteed to be evaluated.

The IterationTag Interface

The IterationTag interface is used to repeatedly reevaluate the body of a custom action. The interface has one method: doAfterBody() which is invoked after each evaluation of the body to determine whether to reevaluate or not.

Reevaluation is requested with the value 2, which in JSP 1.1 is defined to be BodyTag.EVAL_BODY_TAG. That constant value is still kept in JSP 1.2 (for full backwards compatibility) but, to improve clarity, a new name is also available: IterationTag.EVAL_BODY_AGAIN. To stop iterating, the returned value should be 0, which is Tag.SKIP_BODY.

The TagSupport Base Class

The TagSupport class is a base class that can be used when implementing the Tag or IterationTag interfaces.

JSP.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().

Methods

There are two main actions: doStartTag and doEndTag. Once all appropriate properties have been initialized, the doStartTag and doEndTag methods can be invoked on the tag handler. Between these invocations, the tag handler is assumed to hold a state that must be preserved. After the doEndTag invocation, the tag handler is available for further invocations (and it is expected to have retained its properties).

Lifecycle

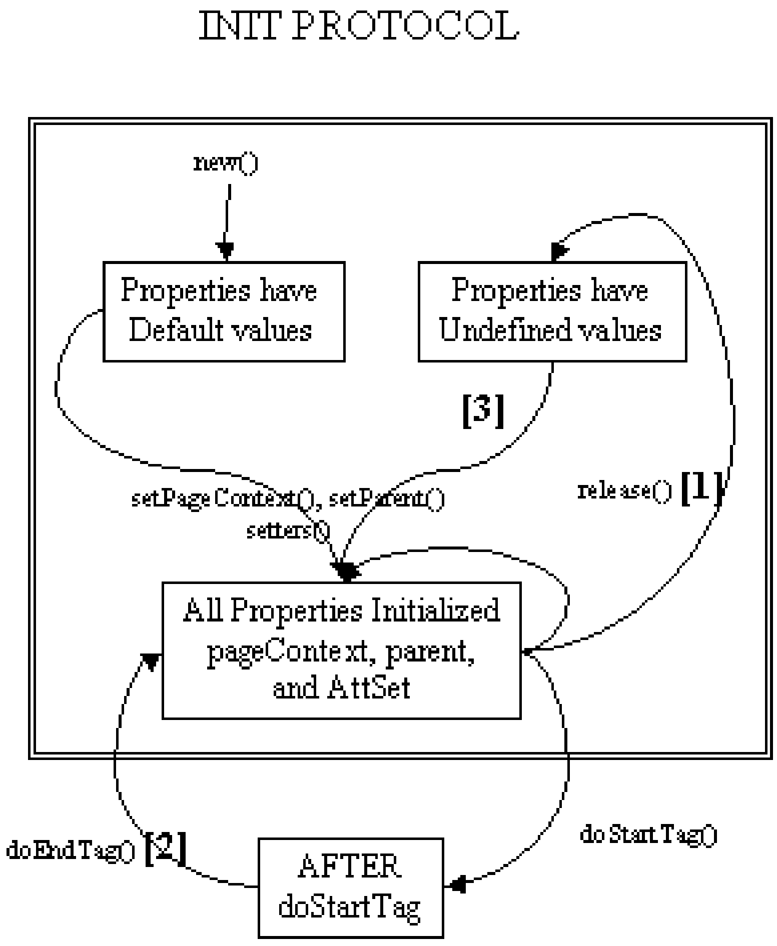
Lifecycle details are described by the transition diagram below, with the following comments:

•[1] This transition is intended to be for releasing long-term data. no guarantees are assumed on whether any properties have been retained or not.

•[2] This transition happens if and only if the tag ends normally without raising an exception

•[3] Note that since there are no guarantees on the state of the properties, a tag handler that had some optional properties set can only be reused if those properties are set to a new (known) value. This means that tag handlers can only be reused within the same "AttSet" (set of attributes that have been set).

•Check the TryCatchFinally interface for additional details related to exception handling and resource management.


INIT PROTOCOL
new()
Properties have Default values
Properties have Undefined values
[3]
setPageContext(), setParent()
setters()
release() [1]
All Properties Initialized pageContext, parent, and AttSet
doEndTag() [2]
doStartTag()
AFTER doStartTag

Once all invocations on the tag handler are completed, the release method is invoked on it. Once a release method is invoked *all* properties, including parent and pageContext, are assumed to have been reset to an unspecified value. The page compiler guarantees that release() will be invoked on the Tag handler before the handler is released to the GC.

Empty and Non-Empty Action

If the TagLibraryDescriptor file indicates that the action must always have an empty action, by an <body-content> entry of “empty”, then the doStartTag() method must return SKIP_BODY. Otherwise, the doStartTag() method may return SKIP_BODY or EVAL_BODY_INCLUDE.

If SKIP_BODY is returned the body, if present, is not evaluated.

If EVAL_BODY_INCLUDE is returned, the body is evaluated and "passed through" to the current out.

JSP.10.1.1.1 Fields

public static final int **EVAL_BODY_INCLUDE**

Evaluate body into existing out stream. Valid return value for doStartTag.

public static final int **EVAL_PAGE**

Continue evaluating the page. Valid return value for doEndTag().

public static final int **SKIP_BODY**

Skip body evaluation. Valid return value for doStartTag and doAfterBody.

public static final int **SKIP_PAGE**

Skip the rest of the page. Valid return value for doEndTag.

JSP.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 doEndTag().

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 Body-Tag.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 doStartTag().

Throws:
JspException., JspException

See Also: BodyTag

public Tag **getParent**()

Get the parent (closest enclosing tag handler) for this tag handler.

The getParent() method can be used to navigate the nested tag handler structure at runtime for cooperation among custom actions; for example, the findAncestorWithClass() method in TagSupport provides a convenient way of doing this.

The current version of the specification only provides one formal way of indicating the observable type of a tag handler: its tag handler implementation class, described in the tag-class subelement of the tag element. This is extended in an informal manner by allowing the tag library author to indicate in the description subelement an observable type. The type should be a subtype of the tag handler implementation class or void. This addititional constraint can be exploited by a specialized container that knows about that specific tag library, as in the case of the JSP standard tag library.

public void **release**()

Called on a Tag handler to release state. The page compiler guarantees that JSP page implementation objects will invoke this method on all tag handlers, but there may be multiple invocations on doStartTag and doEndTag in between.

public void **setPageContext**(PageContext pc)

Set the current page context. This method is invoked by the JSP page implementation object prior to doStartTag().

This value is *not* reset by doEndTag() and must be explicitly reset by a page implementation if it changes between calls to doStartTag().

Parameters:
pc **-** The page context for this tag handler.

public void **setParent**(Tag t)

Set the parent (closest enclosing tag handler) of this tag handler. Invoked by the JSP page implementation object prior to doStartTag().

This value is *not* reset by doEndTag() and must be explicitly reset by a page implementation.

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

JSP.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().

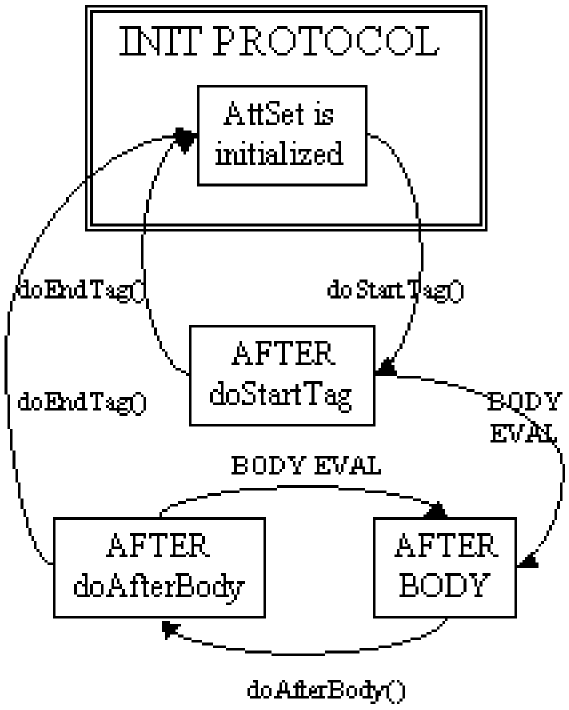
The doAfterBody() method is invoked after every body evaluation to control whether the body will be reevaluated or not. If doAfterBody() returns Iteration-Tag.EVAL_BODY_AGAIN, then the body will be reevaluated. If doAfterBody() returns Tag.SKIP_BODY, then the body will be skipped and doEndTag() will be evaluated instead.

Properties There are no new properties in addition to those in Tag.

Methods There is one new methods: doAfterBody().

Lifecycle

Lifecycle details are described by the transition diagram below. Exceptions that are thrown during the computation of doStartTag(), BODY and doAfterBody() interrupt the execution sequence and are propagated up the stack, unless the tag handler implements the TryCatchFinally interface; see that interface for details.



Empty and Non-Empty Action

If the TagLibraryDescriptor file indicates that the action must always have an empty action, by an <body-content> entry of “empty”, then the doStartTag() method must return SKIP_BODY. Otherwise, the doStartTag() method may return SKIP_BODY or EVAL_BODY_INCLUDE.

If SKIP_BODY is returned the body is not evaluated, and then doEndTag() is invoked.

If EVAL_BODY_INCLUDE is returned, the body is evaluated and “passed through” to the current out, then doAfterBody() is invoked and, after zero or more iterations, doEndTag() is invoked.

JSP.10.1.2.3 Fields

public static final int **EVAL_BODY_AGAIN**

Request the reevaluation of some body. Returned from doAfterBody. For compatibility with JSP 1.1, the value is carefully selected to be the same as the, now deprecated, BodyTag.EVAL_BODY_TAG,

JSP.10.1.2.4 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

JSP.10.1.3 TryCatchFinally

Syntax

public interface TryCatchFinally

Description

The auxiliary interface of a Tag, IterationTag or BodyTag tag handler that wants additional hooks for managing resources.

This interface provides two new methods: doCatch(Throwable) and doFinally(). The prototypical invocation is as follows:

```
h = get a Tag();  // get a tag handler, perhaps from pool
h.setPageContext(pc);  // initialize as desired
h.setParent(null);
h.setFoo("foo");

// tag invocation protocol; see Tag.java
try {
  doStartTag()...
  ....
  doEndTag()...
} catch (Throwable t) {
  // react to exceptional condition
  h.doCatch(t);
} finally {
  // restore data invariants and release per-invocation resources
  h.doFinally();
}

... other invocations perhaps with some new setters
...
h.release();  // release long-term resources
```

JSP.10.1.3.5 Methods

public void **doCatch**(java.lang.Throwable t)

Invoked if a Throwable occurs while evaluating the BODY inside a tag or in any of the following methods: Tag.doStartTag(), Tag.doEndTag(), IterationTag.doAfterBody() and BodyTag.doInitBody().

This method is not invoked if the Throwable occurs during one of the setter methods.

This method may throw an exception (the same or a new one) that will be propagated further the nest chain. If an exception is thrown, doFinally() will be invoked.

This method is intended to be used to respond to an exceptional condition.

Parameters:
t - The throwable exception navigating through this tag.

Throws:
Throwable

public void **doFinally**()

Invoked in all cases after doEndTag() for any class implementing Tag, IterationTag or BodyTag. This method is invoked even if an exception has occurred in the BODY of the tag, or in any of the following methods:

Tag.doStartTag(), Tag.doEndTag(), IterationTag.doAfterBody() and Body-Tag.doInitBody().

This method is not invoked if the Throwable occurs during one of the setter methods.

This method should not throw an Exception.

This method is intended to maintain per-invocation data integrity and resource management actions.

JSP.10.1.4 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 defining new tag handlers implementing Tag.

The TagSupport class is a utility class intended to be used as the base class for new tag handlers. The TagSupport class implements the Tag and IterationTag interfaces and adds additional convenience methods including getter methods for the properties in Tag. TagSupport has one static method that is included to facilitate coordination among cooperating tags.

Many tag handlers will extend TagSupport and only redefine a few methods.

JSP.10.1.4.6 Fields

protected java.lang.String **id**

protected PageContext **pageContext**

JSP.10.1.4.7 Constructors

public **TagSupport**()

Default constructor, all subclasses are required to define only a public constructor with the same signature, and to call the superclass constructor. This constructor is called by the code generated by the JSP translator.

JSP.10.1.4.8 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.

The current version of the specification only provides one formal way of indicating the observable type of a tag handler: its tag handler implementation class, described in the tag-class subelement of the tag element. This is extended in an informal manner by allowing the tag library author to indicate in the description subelement an observable type. The type should be a subtype of the tag handler implementation class or void. This addititional constraint can be exploited by a specialized container that knows about that specific tag library, as in the case of the JSP standard tag library.

When a tag library author provides information on the observable type of a tag handler, client programmatic code should adhere to that constraint. Specifically, the Class passed to findAncestorWithClass should be a subtype of the observable type.

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

klass - The subclass of Tag or interface to be matched

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.

JSP.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 BodyTag-Support class provides a useful base class to simplify writing these handlers.

If a Tag handler wants to have access to the content of its body then it must implement the BodyTag interface. This interface extends IterationTag, provides two additional methods setBodyContent(BodyContent) and doInitBody() and refers to an object of type BodyContent.

A BodyContent is a subclass of JspWriter that has a few additional methods to convert its contents into a String, insert the contents into another JspWriter, to get a Reader into its contents, and to clear the contents. Its semantics also assure that buffer size will never be exceeded.

The JSP page implementation will create a BodyContent if the doStartTag() method returns a EVAL_BODY_BUFFERED. This object will be passed to doInitBody(); then the body of the tag will be evaluated, and *during that evaluation* ***out will be bound to the BodyContent just passed to the BodyTag handler***. Then doAfterBody() will be evaluated. If that method returns SKIP_BODY, no more evaluations of the body will be done; if the method returns EVAL_BODY_AGAIN, then the body will be evaluated, and doAfterBody() will be invoked again.

The content of a BodyContent instance remains available until after the invocation of its associated doEndBody() method.

A common use of the BodyContent is to extract its contents into a String and then use the String as a value for some operation. Another common use is to take its contents and push it into the out Stream that was valid when the start tag was encountered (that is available from the PageContext object passed to the handler in setPageContext).

JSP.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.

A BodyContent is made available to a BodyTag through a setBodyContent() call. The tag handler can use the object until after the call to doEndTag().

JSP.10.2.1.9 Constructors

protected **BodyContent**(JspWriter e)

Protected constructor. Unbounded buffer, no autoflushing.

JSP.10.2.1.10 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

JSP.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 (by code generated by the JSP compiler) to capture the body evaluation. The code generated by the JSP compiler obtains the BodyContent object by calling the pushBody method of the current pageContext, which additionally has the effect of saving the previous out value. The page compiler returns this object by calling the popBody method of the PageContext class; the call also restores the value of out.

The interface provides one new property with a setter method and one new action method.

Properties

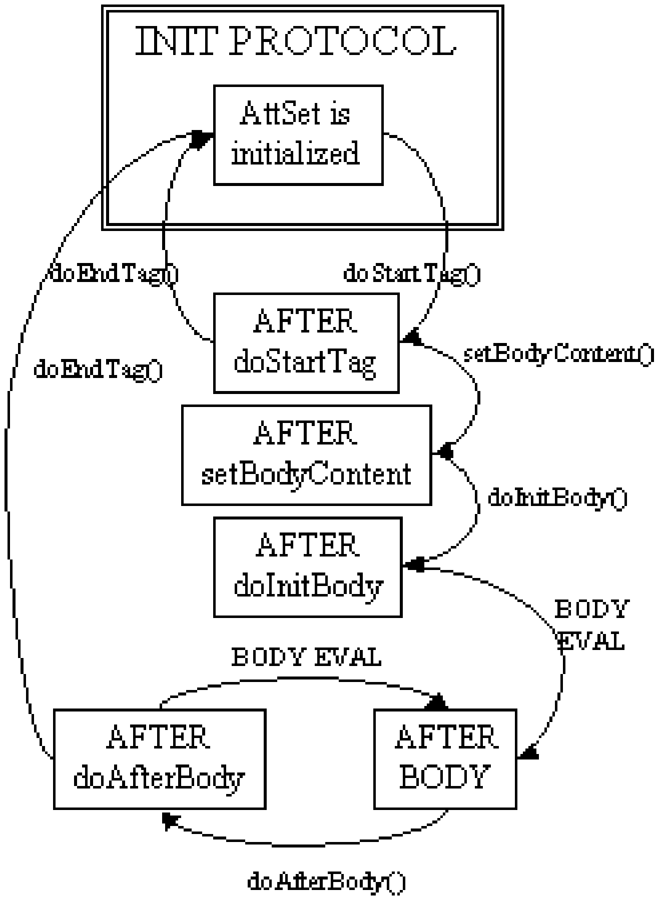
There is a new property: bodyContent, to contain the BodyContent object, where the JSP Page implementation object will place the evaluation (and reevaluation, if appropriate) of the body. The setter method (setBodyContent) will only be invoked if doStartTag() returns EVAL_BODY_BUFFERED.

Methods

In addition to the setter method for the bodyContent property, there is a new action methods: doInitBody(), which is invoked right after setBodyContent() and before the body evaluation. This method is only invoked if doStartTag() returns EVAL_BODY_BUFFERED.

Lifecycle

Lifecycle details are described by the transition diagram below. Exceptions that are thrown during the computation of doStartTag(), setBodyContent(), doInitBody(), BODY, doAfterBody() interrupt the execution sequence and are propagated up the stack, unless the tag handler implements the TryCatchFinally interface; see that interface for details.



Empty and Non-Empty Action

If the TagLibraryDescriptor file indicates that the action must always have an empty action, by an <body-content> entry of “empty”, then the doStartTag() method must return SKIP_BODY. Otherwise, the doStartTag() method may return SKIP_BODY, EVAL_BODY_INCLUDE, or EVAL_BODY_BUFFERED.

If SKIP_BODY is returned the body is not evaluated, and doEndTag() is invoked.

If EVAL_BODY_INCLUDE is returned, setBodyContent() is not invoked, doInitBody() is not invoked, the body is evaluated and “passed through” to the current out, doAfterBody() is invoked and then, after zero or more iterations, doEndTag() is invoked.

If EVAL_BODY_BUFFERED is returned, setBodyContent() is invoked, doInit-Body() is invoked, the body is evaluated, doAfterBody() is invoked, and then, after zero or more iterations, doEndTag() is invoked.

JSP.10.2.2.11 Fields

public static final int **EVAL_BODY_BUFFERED**

Request the creation of new buffer, a BodyContent on which to evaluate the body of this tag. Returned from doStartTag when it implements BodyTag. This is an illegal return value for doStartTag when the class does not implement BodyTag.

public static final int **EVAL_BODY_TAG**

Deprecated. As of Java JSP API 1.2, use BodyTag.EVAL_BODY_BUFFERED or IterationTag.EVAL_BODY_AGAIN.

Deprecated constant that has the same value as EVAL_BODY_BUFFERED and EVAL_BODY_AGAIN. This name has been marked as deprecated to encourage the use of the two different terms, which are much more descriptive.

JSP.10.2.2.12 Methods

public void **doInitBody**()

Prepare for evaluation of the body. This method is invoked by the JSP page implementation object after setBodyContent and before the first time the body is to be evaluated. This method will not be invoked for empty tags or for non-empty tags whose doStartTag() method returns SKIP_BODY or EVAL_BODY_INCLUDE.

The JSP container will resynchronize 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. This method will be invoked before doInitBody. This method will not be invoked for empty tags or for non-empty tags whose doStartTag() method returns SKIP_BODY or EVAL_BODY_INCLUDE.

When setBodyContent is invoked, the value of the implicit object out has already been changed in the pageContext object. The BodyContent object passed will have not data on it but may have been reused (and cleared) from some previous invocation.

The BodyContent object is available and with the appropriate content until after the invocation of the doEndTag method, at which case it may be reused.

Parameters:
b - the BodyContent

JSP.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 BodyTagSupport and only redefine a few methods.

JSP.10.2.3.13 Fields

protected BodyContent **bodyContent**

JSP.10.2.3.14 Constructors

public **BodyTagSupport**()

Default constructor, all subclasses are required to only define a public constructor with the same signature, and to call the superclass constructor. This constructor is called by the code generated by the JSP translator.

JSP.10.2.3.15 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_BUFFERED

Overrides: public int doStartTag() in class TagSupport

Returns: EVAL_BODY_BUFFERED;

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

JSP.10.3 Annotated Tag Handler Management Example

Below is a somewhat complete example of the way one JSP container could choose to do some tag handler management. There are many other strategies that could be followed, with different pay offs.

The example is as below. In this example, we are assuming that x:iterate is an iterative tag, while x:doit and x:foobar are simple tag. We will also assume that x:iterate and x:foobar implement the TryCatchFinally interface, while x:doit does not.

```
<x:iterate src=“foo”>
  <x:doit att1=“one” att2=“<%= 1 + 1 %>” />
  <x:foobar />
  <x:doit att1=“one” att2=“<%= 2 + 2 %>” />
</x:iterate>
<x:doit att1=“one” att2=“<%= 3 + 3 %>” />
```

The particular code shown below assumes there is some pool of tag handlers that are managed (details not described, although pool managing is simpler when there are no optional attributes), and attemps to reuse tag handlers if possible. The code also “hoists” setting of properties to reduce the cost when appropriate, e.g. inside an iteration.

```
boolean b1, b2;
IterationTag i; // for x:iterate
Tag d; // for x:doit
Tag d; // for x:foobar
page: // label to end of page...
// initialize iteration tag
i = get tag from pool or new();
i.setPageContext(pc);
i.setParent(null);
i.setSrc("foo");
// x:iterate implements TryCatchFinally
try {
    if ((b1 = i.doStartTag()) == EVAL_BODY_INCLUDE) {
        // initialize doit tag
        // code has been moved out of the loop for show
        d = get tag from pool or new();
        d.setPageContext(pc);
        d.setParent(i);
        d.setAtt1("one");
    loop:
        while (1) do {
            // I'm ignoring newlines...
            // two invocations, fused together
            // first invocation of x:doit
            d.setAtt2(1+1);
            if ((b2 = d.doStartTag()) == EVAL_BODY_INCLUDE) {
                // nothing
            } else if (b2 != SKIP_BODY) {
                // Q? protocol error ...
            }
            if ((b2 = d.doEndTag()) == SKIP_PAGE) {
                break page;  // be done with it.
            } else if (b2 != EVAL_PAGE) {
                // Q? protocol error
            }
    // x:foobar invocation
            f = get tag from pool or new();
            f.setPageContext(pc);
            f.setParent(i);
            // x:foobar implements TryCatchFinally
            try {

                if ((b2 = f.doStartTag()) == EVAL_BODY_INCLUDE) {
                    // nothing
                } else if (b2 != SKIP_BODY) {
                    // Q? protocol error
                }
                if ((b2 = f.doEndTag()) == SKIP_PAGE) {
                    break page;  // be done with it.
                } else if (b2 != EVAL_PAGE) {
                    // Q? protocol error
                }
            } catch (Throwable t) {
                f.doCatch(t); // caught, may been rethrown!
            } finally {
                f.doFinally();
            }
            // put f back to pool

            // second invocation of x:doit
            d.setAtt2(2+2);
            if ((b2 = d.doStartTag()) == EVAL_BODY_INCLUDE) {
                // nothing
            } else if (b2 != SKIP_BODY) {
                // Q? protocol error
            }
            if ((b2 = d.doEndTag()) == SKIP_PAGE) {
                break page;  // be done with it.
            } else if (b2 != EVAL_PAGE) {
                // Q? protocol error
            }
            if ((b2 = i.doAfterBody()) == EVAL_BODY_AGAIN) {
                break loop;
            } else if (b2 != SKIP_BODY) {
                // Q? protocol error
            }
        // loop
        }
    } else if (b1 != SKIP_BODY) {
        // Q? protocol error
    }
    // tail end of the IteratorTag ...
    if ((b1 = i.doEndTag()) == SKIP_PAGE) {
        break page;   // be done with it.
    } else if (b1 != EVAL_PAGE) {
        // Q? protocol error
    }

    // third invocation
    // this tag handler could be reused from the previous ones.
    d = get tag from pool or new();
    d.setPageContext(pc);
    d.setParent(null);
    d.setAtt1(“one”);
    d.setAtt2(3+3);
    if ((b1 = d.doStartTag()) == EVAL_BODY_INCLUDE) {
        // nothing
    } else if (b1 != SKIP_BODY) {
        // Q? protocol error
    }
    if ((b1 = d.doEndTag()) == SKIP_PAGE) {
        break page;  // be done with it.
    } else if (b1 != EVAL_PAGE) {
        // Q? protocol error
    }
} catch (Throwable t) {
    i.doCatch(t); // caught, may been rethrown!
} finally {
    i.doFinally();
}
```

JSP.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.

JSP.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 property settings and then follow the protocol described by the interfaces Tag, IterationTag, BodyTag, and TryCatchFinally. The implementation guarantees that all tag handler instances are initialized and all are released, but the implementation can assume that previous settings are preserved by a tag handler, to reduce run-time costs.

Scripting Variables

JSP supports scripting variables that can be declared within a scriptlet and can be used in another. JSP actions also can be used to define scripting variables so they can used in scripting elements, or in other actions. This is very useful in some cases; for example, the jsp:useBean standard action may define an object which can later be used through a scripting variable.

In some cases the information on scripting variables can be described directly into the TLD using elements. A special case is typical interpretation of the &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. In some cases, additional request-time validation will be done dynamically within the methods in the Tag instance. If an error is discovered, an instance of JspTagException can be thrown. If uncaught, this object will invoke the errorpage mechanism of JSP.

The TagLibraryValidator is an addition to the JSP 1.2 specification and is very open ended, being strictly more powerful than the TagExtraInfo mechanism. A JSP page is presented via the PageData object, which abstracts the XML view of the JSP page.

A PageData instance will provides an InputStream (read-only) on the page. Later specifications may add other views on the page (DOM, SAX, JDOM are all candidates), for now these views can be generated from the InputStream and perhaps can be cached for improved performance (recall the view of the page is just read-only).

A JSP container may optionally support a jsp:id attribute to provide higher quality validation errors. When supported, the container will track the JSP pages as passed to the container, and will assign to each element a unique “id”, which is passed as the value of the jsp:id attribute. Each XML element in the XML view available will be extended with this attribute. The TagLibraryValidator can then use the attribute in one or more ValidationMessage objects. The container then, in turn, can use these values to provide more precise information on the location of an error.

Validation Details

In detail, validation is done as follows:

First, the JSP page is parsed using the information in the TLD. At this stage valid mandatory and optional attributes are checked.

Second, for all taglib directives in the page, and in the lexical order in which they appear, their associated validator class (if any) is invoked. This involves several substeps.

The first substep is to obtain an initialized validator instance by either:

- •construct a new instance and invoke setInitParameters() on it, or
- •obtain an existing instance that is not being used, invoke release() on it, and then invoke setInitParameters() on it, or
- •locate an existing instance that is not being used on which the desired set-InitParameters() has already been invoked

The class name is as indicated in the <validator-class> element, and the Map passed through setInitParameters() is as described in the <init-params> element. All TagLibraryValidator classes are supposed to keep their initParameters until new ones are set, or until release() is invoked on them.

The second substep is to perform the actual validation. This is done by invoking the validate() method with a prefix, uri, and PageData that correspond to the taglib directive instance being validated and the PageData representing the page.

The last substep is to invoke the release() method on the validator tag when it is no longer needed. This method releases all resources.

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

JSP.10.5.1 TagLibraryInfo

Syntax

public abstract class TagLibraryInfo

Description

Translation-time information associated with a taglib directive, and its underlying TLD file. Most of the information is directly from the TLD, except for the prefix and the uri values used in the taglib directive

JSP.10.5.1.16 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**

JSP.10.5.1.17 Constructors

protected **TagLibraryInfo**(java.lang.String prefix, java.lang.String uri)

Constructor. This will invoke the constructors for TagInfo, and TagAttributeInfo after parsing the TLD file.

Parameters:
prefix - the prefix actually used by the taglib directive

uri - the URI actually used by the taglib directive

JSP.10.5.1.18 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 a taglib 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 preferred short name (prefix) as indicated in the TLD. This may be used by authoring tools as the preferred prefix to use when creating an 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.

JSP.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.

JSP.10.5.2.19 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

JSP.10.5.2.20 Constructors

public **TagInfo**(java.lang.String tagName, java.lang.String tagClassName, java.lang.String bodycontent, java.lang.String infoString, TagLibraryInfo taglib, TagExtraInfo tagExtraInfo, TagAttributeInfo[] attributeInfo)

Constructor for TagInfo from data in the JSP 1.1 format for TLD. This class is to be instantiated only from the TagLibrary code under request from some JSP code that is parsing a TLD (Tag Library Descriptor). Note that, since TagLibibraryInfo reflects both TLD information and taglib directive information, a TagInfo instance is dependent on a taglib directive. This is probably a design error, which may be fixed in the future.

Parameters:
tagName - The name of this tag

tagClassName - The name of the tag handler class

bodycontent - Information on the body content of these tags

infoString - The (optional) string information for this tag

taglib - The instance of the tag library that contains us.

tagExtraInfo - The instance providing extra Tag info. May be null

attributeInfo - An array of AttributeInfo data from descriptor. May be null;

public **TagInfo**(java.lang.String tagName, java.lang.String tagClassName, java.lang.String bodycontent, java.lang.String infoString, TagLibraryInfo taglib, TagExtraInfo tagExtraInfo, TagAttributeInfo[] attributeInfo, java.lang.String displayName, java.lang.String smallIcon, java.lang.String largeIcon, TagVariableInfo[] tvi)

Constructor for TagInfo from data in the JSP 1.2 format for TLD. This class is to be instantiated only from the TagLibrary code under request from some JSP code that is parsing a TLD (Tag Library Descriptor). Note that, since TagLibibraryInfo reflects both TLD information and taglib directive information, a TagInfo instance is dependent on a taglib directive. This is probably a design error, which may be fixed in the future.

Parameters:
tagName - The name of this tag

tagClassName - The name of the tag handler class

bodycontent - Information on the body content of these tags

infoString - The (optional) string information for this tag

taglib - The instance of the tag library that contains us.

tagExtraInfo - The instance providing extra Tag info. May be null

attributeInfo - An array of AttributeInfo data from descriptor. May be null;

displayName - A short name to be displayed by tools

smallIcon - Path to a small icon to be displayed by tools

largeIcon - Path to a large icon to be displayed by tools

tagVariableInfo - An array of a TagVariableInfo (or null)

JSP.10.5.2.21 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 **getDisplayName**()

Get the displayName

Returns: A short name to be displayed by tools

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

The information string for the tag.

Returns: the info string

public java.lang.String **getLargeIcon**()

Get the path to the large icon

Returns: Path to a large icon to be displayed by tools public java.lang.String **getSmallIcon**()

Get the path to the small icon

Returns: Path to a small icon to be displayed by tools

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 TagVariableInfo[] **getTagVariableInfos**()

Get TagVariableInfo objects associated with this TagInfo

Returns: A TagVariableInfo object associated with this

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 void **setTagExtraInfo**(TagExtraInfo tei)

Set the instance for extra tag information

Parameters:
tei - the TagExtraInfo instance

public void **setTagLibrary**(TagLibraryInfo tl)

Set the TagLibraryInfo property. Note that a TagLibraryInfo element is dependent not just on the TLD information but also on the specific taglib instance used. This means that a fair amount of work needs to be done to construct and initialize TagLib objects. If used carefully, this setter can be used to avoid having to create new TagInfo elements for each taglib directive.

Parameters:
tl - the TagLibraryInfo to assign

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

Stringify for debug purposes...

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

JSP.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.

JSP.10.5.3.22 Fields

public static final java.lang.String **ID**

“id” is wired in to be ID. There is no real benefit in having it be something else IDREFs are not handled any differently.

JSP.10.5.3.23 Constructors

public **TagAttributeInfo**(java.lang.String name, boolean required,
java.lang.String type, boolean reqTime)

Constructor for TagAttributeInfo. This class is to be instantiated only from the TagLibrary code under request from some JSP code that is parsing a TLD (Tag Library Descriptor).

Parameters:
name - The name of the attribute.

required - If this attribute is required in tag instances.

type - The name of the type of the attribute.

reqTime - Whether this attribute holds a request-time Attribute.

JSP.10.5.3.24 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

JSP.10.5.4 PageData

Syntax

public abstract class PageData

Description

Translation-time information on a JSP page. The information corresponds to the XML view of the JSP page.

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

JSP.10.5.4.25 Constructors

public **PageData**()

JSP.10.5.4.26 Methods

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

Returns an input stream on the XML view of a JSP page. Recall tht the XML view of a JSP page has the include directives expanded.

Returns: An input stream on the document.

JSP.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.

The TLD file associates a TagLibraryValidator class and some init arguments with a tag library.

The JSP container is reponsible for locating an appropriate instance of the appropriate subclass by

- new a fresh instance, or reuse an available one
- invoke the setInitParams(Map) method on the instance

once initialized, the validate(String, String, PageData) method will be invoked, where the first two arguments are the prefix and uri arguments used in the taglib directive.

A TagLibraryValidator instance may create auxiliary objects internally to perform the validation (e.g. an XSchema validator) and may reuse it for all the pages in a given translation run.

The JSP container is not guaranteed to serialize invocations of validate() method, and TagLibraryValidators should perform any synchronization they may require.

A JSP container may optionally support a jsp:id attribute to provide higher quality validation errors. When supported, the container will track the JSP pages as passed to the container, and will assign to each element a unique "id", which is passed as the value of the jsp:id attribute. Each XML element in the XML view available will be extended with this attribute. The TagLibraryValidator can then use the attribute in one or more ValidationMessage objects. The container then, in turn, can use these values to provide more precise information on the location of an error.

JSP.10.5.5.27 Constructors

public **TagLibraryValidator**()

JSP.10.5.5.28 Methods

public java.util.Map **getInitParameters**()

Get the init parameters data as an immutable Map. Parameter names are keys, and parameter values are the values.

Returns: The init parameters as an immutable map.

public void **release**()

Release any data kept by this instance for validation purposes

public void **setInitParameters**(java.util.Map map)

Set the init data in the TLD for this validator. Parameter names are keys, and parameter values are the values.

Parameters:
initMap - A Map describing the init parameters

public ValidationMessage[] **validate**(java.lang.String prefix, java.lang.String uri, PageData page)

Validate a JSP page. This will get invoked once per directive in the JSP page. This method will return null if the page is valid; otherwise the method should return an array of ValidationMessage objects. An array of length zero is also interpreted as no errors.

Parameters:
prefix - the value of the prefix argument in the directive

uri - the value of the uri argument in the directive

thePage - the JspData page object

Returns: A null object, or zero length array if no errors, an array of ValidationMessages otherwise.

JSP.10.5.6 ValidationMessage

Syntax
public class ValidationMessage

Description

A validation message from a TagLibraryValidator.

A JSP container may (optionally) support a jsp:id attribute to provide higher quality validation errors. When supported, the container will track the JSP pages as passed to the container, and will assign to each element a unique "id", which is passed as the value of the jsp:id attribute. Each XML element in the XML view available will be extended with this attribute. The TagLibraryValidator can then use the attribute in one or more ValidationMessage objects. The container then, in turn, can use these values to provide more precise information on the location of an error.

JSP.10.5.6.29 Constructors

public **ValidationMessage**(java.lang.String id, java.lang.String message)

Create a ValidationMessage. The message String should be non-null. The value of id may be null, if the message is not specific to any XML element, or if no jsp:id attributes were passed on. If non-null, the value of id must be the value of a jsp:id attribute for the PageData passed into the validate() method.

Parameters:
id - Either null, or the value of a jsp:id attribute.

message - A localized validation message.

JSP.10.5.6.30 Methods

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

Get the jsp:id. Null means that there is no information available.

Returns: The jsp:id information.

public java.lang.String **getMessage**()

Get the localized validation message.

Returns: A validation message

JSP.10.5.7 TagExtraInfo

Syntax

public abstract class TagExtraInfo

Description

Optional class provided by the tag library author to describe additional translation-time information not described in the TLD. The TagExtraInfo class is mentioned in the Tag Library Descriptor file (TLD).

This class can be used:
•to indicate that the tag defines scripting variables
•to perform translation-time validation of the tag attributes.

It is the responsibility of the JSP translator that the initial value to be returned by calls to getTagInfo() corresponds to a TagInfo object for the tag being translated. If an explicit call to setTagInfo() is done, then the object passed will be returned in subsequent calls to getTagInfo().

The only way to affect the value returned by getTagInfo() is through a setTagInfo() call, and thus, TagExtraInfo.setTagInfo() is to be called by the JSP translator, with a TagInfo object that corresponds to the tag being translated. The call should happen before any invocation on isValid() and before any invocation on getVariableInfo().

JSP.10.5.7.31 Constructors

public **TagExtraInfo**()

JSP.10.5.7.32 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

JSP.10.5.8 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 TagExtraInfo, which are invoked at translation time.

JSP.10.5.8.33 Fields

public static final java.lang.Object **REQUEST_TIME_VALUE**

Distinguished value for an attribute to indicate its value is a request-time expression (which is not yet available because TagData instances are used at translation-time).

JSP.10.5.8.34 Constructors

public **TagData**(java.util.Hashtable attrs)

Constructor for a TagData. If you already have the attributes in a hashtable, use this constructor.

Parameters:
attrs - A hashtable to get the values from.

public **TagData**(java.lang.Object[][] atts)

Constructor for TagData.

A typical constructor may be

```
static final Object[][] att = {{“connection”, “conn0”},
{“id”, “query0”}};
static final TagData td = new TagData(att);
```

All values must be Strings except for those holding the distinguished object REQUEST_TIME_VALUE.

Parameters:
atts - the static attribute and values. May be null.

JSP.10.5.8.35 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.

JSP.10.5.9 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 values of 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.

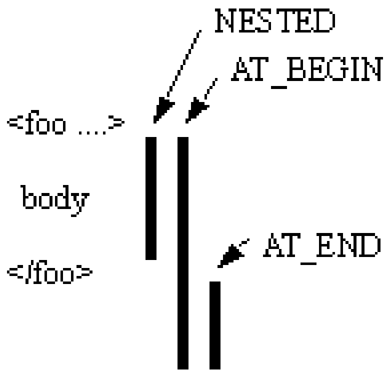
Usage Comments

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 most other remaining cases it will refer to a class that is in the platform on which the JSP processor is built (like J2EE). Using fully qualified class names in this manner makes the usage relatively resistant to configuration errors.

A short name is usually generated by the tag library based on some attributes passed through from the custom action user (the author), and it is thus less robust: for instance a missing import directive in the referring JSP page will lead to an invalid short name class and a translation error.

Synchronization Protocol

The result of the invocation on getVariableInfo is an array of VariableInfo objects. Each such object describes a scripting variable by providing its name, its type, whether the variable is new or not, and what its scope is. Scope is best described through a picture:



The JSP 1.2 specification defines the interpretation of 3 values:

•NESTED, if the scripting variable is available between the start tag and the end tag of the action that defines it.

•AT_BEGIN, if the scripting variable is available from the start tag of the action that defines it until the end of the scope.

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

The scope value for a variable implies what methods may affect its value and thus where synchronization is needed:

•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.

JSP.10.5.9.36 Fields

public static final int **AT_BEGIN**

Scope information that scripting variable is visible after start tag

public static final int **AT_END**

Scope information that scripting variable is visible after end tag

public static final int **NESTED**

Scope information that scripting variable is visible only within the start/end tags

JSP.10.5.9.37 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 Tag-ExtraInfo 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

JSP.10.5.9.38 Methods

public java.lang.String **getClassName**()

public boolean **getDeclare**()

public int **getScope**()

public java.lang.String **getVarName**()

JSP.10.5.10 TagVariableInfo

Syntax
public class TagVariableInfo

Description

Variable information for a tag in a Tag Library; This class is instantiated from the Tag Library Descriptor file (TLD) and is available only at translation time. This object should be immutable. This information is only available in JSP 1.2 format

JSP.10.5.10.39 Constructors

public **TagVariableInfo**(java.lang.String nameGiven,
java.lang.String nameFromAttribute, java.lang.String className,
boolean declare, int scope)

Constructor for TagVariableInfo

Parameters:
nameGiven - value of <name-given>

nameFromAttribute - value of <name-from-attribute>

className - value of <variable-class>

declare - value of <declare>

scope - value of <scope>

JSP.10.5.10.40 Methods

public java.lang.String **getClassName**()

The body of the <variable-class> element.

Returns: The name of the class of the variable

public boolean **getDeclare**()

The body of the <declare> element

Returns: Whether the variable is to be declared or not

public java.lang.String **getNameFromAttribute**()

The body of the <name-from-attribute> element. This is the name of an attribute whose (translation-time) value will give the name of the variable. One of <name-given> or <name-from-attribute> is required.

Returns: The attribute whose value defines the variable name

public java.lang.String **getNameGiven**()

The body of the <name-given> element

Returns: The variable name as a constant

public int **getScope**()

The body of the <scope> element

Returns: The scope to give the variable.

APPENDIX JSP.A
Packaging JSP Pages

This appendix shows two simple examples of packaging a JSP page into a WAR for delivery into a Web container. In the first example, the JSP page is delivered in source form. This is likely to be the most common example. In the second example the JSP page is compiled into a Servlet that uses only Servlet 2.3 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 than 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 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.2 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 webapp
    SYSTEM "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
<webapp>
    <session-config>
        <session-timeout> 1 </session-timeout>
    </session-config>
</webapp>
```

A.3 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.4 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_2_3.dtd">
<webapp>
    <servlet>
        <servlet-name> HelloWorld </servlet-name>
        <servlet-class> _jsp_HelloWorld_XXX_Impl.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.5 The WAR for the compiled JSP page

Finally everything is packaged together into a WAR:

```
/WEB-INF/web.xml

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

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

APPENDIX JSP.B

DTD and Schemas for XML Syntax

This appendix includes the DTD and XSchema for JSP pages in XML syntax. This appendix is non-normative.

As indicated in Section JSP.5.4, a DTD is not a good description to be used for validating a document that is using namespaces like a JSP page in XML syntax, but its familiarity with many readers makes it nevertheless useful.

B.1 DTD for JSP documents

The following is a DTD for JSP documents. Since a JSP document is a namespace-aware document, this DTD is included here just for documentation purposes.

```
<!-- DTD for JSP 1.2 -->

<!--
    This DTD is not conditional on any parameter entities in the
    internal subset and does not export any general entities.
-->



<!-- Constrained Names -->
```

```
<!ENTITY % URI “CDATA”>
   <!-- a Uniform Resource Identifier, see [RFC2396] -->

<!ENTITY % UriList “CDATA”>
   <!-- a space separated list of Uniform Resource Identifiers -->

<!ENTITY % URL “CDATA”>
   <!-- a relative urlSpec is as in Section 2.10.2. -->

<!ENTITY % BeanID “IDREF”>
   <!-- a previously declared bean ID in the current scope. -->

<!ENTITY % Prefix “CDATA”>
   <!-- a Name that contains no : characters. -->

<!ENTITY % ClassName “CDATA”>
   <!-- a fully qualified class name. -->

<!ENTITY % TypeName “CDATA”>
   <!-- a fully qualified class or interface name. -->

<!ENTITY % BeanName “CDATA”>
   <!-- a bean name as expected by java.beans.Beans instantiate(). -->

<!ENTITY % Content “CDATA”>
   <!-- a MIME type followed by an IANA charset, as “ type [; S? [‘charset=’] char-
set] “ -->

<!ENTITY % Length “CDATA”>
   <!-- nn for pixels or nn% for percentage length -->

<!ENTITY % Pixels “CDATA”>
   <!-- integer representing length in pixels -->

<!ENTITY % Bool “(true|false|yes|no)”>
   <!-- boolean -->

<!-- used for object, applet, img, input and iframe -->
<!ENTITY % ImgAlign “(top|middle|bottom|left|right)”>

<!-- Element Groups -->
```

```
<!ENTITY % Directives “jsp:directive.page|jsp:directive.include”>

<!ENTITY % Scripts “jsp:scriptlet|jsp:declaration|jsp:expression”>

<!ENTITY % Actions
   “jsp:useBean
   |jsp:setProperty
   |jsp:getProperty
   |jsp:include
   |jsp:forward
   |jsp:plugin”
>

<!ENTITY % Body “(jsp:text|%Directives;|%Scripts;|%Actions;)*”>


<!-- Elements  -->

<!--   Root element of a JSP page.
-->
<!ELEMENT jsp:root %Body;>
<!ATTLIST jsp:root
   xmlns:jsp      CDATA          “http://java.sun.com/JSP/Page”
   version        CDATA          #REQUIRED
>

<!ELEMENT jsp:directive.page EMPTY>
<!ATTLIST jsp:directive.page
   language       CDATA          “java”
   extends        %ClassName;    #IMPLIED
   contentType    %Content;      “text/html; ISO-8859-1”
   import         CDATA          #IMPLIED
   session        %Bool;         “true”
   buffer         CDATA          “8kb”
   autoFlush      %Bool;         “true”
   isThreadSafe   %Bool;         “true”
   info           CDATA          #IMPLIED
   errorPage      %URL;          #IMPLIED
   isErrorPage    %Bool;         “false”
>
```

```
<!-- the jsp:directive.include element only appears in JSP documents and does
not appear in XML views of JSP pages -->

<!ELEMENT jsp:directive.include EMPTY>
<!ATTLIST jsp:directive.include
   file         %URI;         #REQUIRED
>

<!ELEMENT jsp:scriptlet (#PCDATA)>

<!ELEMENT jsp:declaration (#PCDATA)>

<!ELEMENT jsp:expression (#PCDATA)>

<!ELEMENT jsp:useBean %Body;>
<!ATTLIST jsp:useBean
   id           ID            #REQUIRED
   class        %ClassName;   #IMPLIED
   type         %TypeName;    #IMPLIED
   beanName     %BeanName;    #IMPLIED
   scope        (page
                |session
                |request
                |application)  "page"
>

<!ELEMENT jsp:setProperty EMPTY>
<!ATTLIST jsp:setProperty
   name         %BeanID;      #REQUIRED
   property     CDATA         #REQUIRED
   value        CDATA         #IMPLIED
   param        CDATA         #IMPLIED
>

<!ELEMENT jsp:getProperty EMPTY>
<!ATTLIST jsp:getProperty
   name         %BeanID;      #REQUIRED
   property     CDATA         #REQUIRED
>
```

```
<!ELEMENT jsp:include (jsp:param*)>
<!ATTLIST jsp:include
   flush          %Bool;         “false”
   page           %URL;          #REQUIRED
>

<!ELEMENT jsp:forward (jsp:param*)>
<!ATTLIST jsp:forward
   page           %URL;          #REQUIRED
>

<!ELEMENT jsp:plugin (jsp:params?, jsp:fallback?)>
<!ATTLIST jsp:plugin
   type           (bean|applet)  #REQUIRED
   code           %URI;          #IMPLIED
   codebase       %URI;          #IMPLIED
   align          %ImgAlign;     #IMPLIED
   archive        %UriList;      #IMPLIED
   height         %Length;       #IMPLIED
   hspace         %Pixels;       #IMPLIED
   jreversion     CDATA          “1.2”
   name           NMTOKEN        #IMPLIED
   vspace         %Pixels;       #IMPLIED
   width          %Length;       #IMPLIED
   nspluginurl    %URI;          #IMPLIED
   iepluginurl    %URI;          #IMPLIED
>

<!ELEMENT jsp:params (jsp:param+)>

<!ELEMENT jsp:param EMPTY>
<!ATTLIST jsp:param
   name           CDATA          #REQUIRED
   value          CDATA          #REQUIRED
>

<!ELEMENT jsp:text #PCDATA>
```

B.2 XSchema Description of JSP documents

The following is a description using XML Schema:

```
<?xml version ="1.0"?>
<!DOCTYPE schema [
<!-- Patterns -->
<!ENTITY Identifier   "(\p{L}|_|$)(\p{N}|\p{L}|_|$)*">
<!ENTITY TypeName     "&Identifier;(\.&Identifier;)*">
<!ENTITY WS       "\s*">
<!ENTITY Import     "&TypeName;(\.\*)?">
<!ENTITY ImportList   "&Import;(&WS;,&WS;&Import;)*">
<!ENTITY SetProp    "(&Identifier;|\*)">
<!ENTITY RelativeURL  "[^:#/\?]*(:{0,0}|[#/\?].*)">
<!ENTITY Length     "[0-9]*&#x25;?">
<!ENTITY AsciiName    "[A-Za-z0-9_-]*">
<!ENTITY ValidContentType
              "&AsciiName;/&AsciiName;(;&WS;(charset=)?&AsciiName;)?">
<!ENTITY ValidPageEncoding  "&AsciiName;/&AsciiName;">
<!ENTITY Buffer     "[0-9]+kb">
<!ENTITY RTexpr     "&#x25;=.*&#x25;">
]>


<!--Conforms to w3c http://www.w3.org/2001/XMLSchema -->

<xsd:schema
   xmlns = "http://java.sun.com/JSP/Page"
   xmlns:xsd = "http://www.w3.org/2001/XMLSchema"
   xmlns:jsp = "http://java.sun.com/JSP/Page"
   targetNamespace = "http://java.sun.com/JSP/Page"
   elementFormDefault = "qualified"
   attributeFormDefault = "unqualified">

  <xsd:annotation>
   <xsd:documentation>
     XML Schema for JSP 1.2.

     This schema is based upon the recent (May 5th, 2001)
     W3C recommendation for XML Schema.

     A JSP translator should reject an XML-format file that is
     not strictly valid according to this schema or does not observe
     the constraints documented here. A translator is not required
     to use this schema for validation or to use a validating parser.
   </xsd:documentation>
  </xsd:annotation>
```

```
 <!-- Complex Types -->

 <xsd:complexType name = "Body">
  <xsd:annotation>
   <xsd:documentation>
    Body defines the "top-level" elements in root and beanInfo.
    There are probably other elements that should use it.
   </xsd:documentation>
  </xsd:annotation>
  <xsd:group ref = "Bodygroup" minOccurs = "0" maxOccurs = "unbounded"/>
 </xsd:complexType>

 <xsd:complexType name = "BasicType">
  <xsd:simpleContent>
   <xsd:extension base = "xsd:string">
     <xsd:attribute ref = "jsp:id"/>
   </xsd:extension>
  </xsd:simpleContent>
 </xsd:complexType>

 <!-- groups -->

 <xsd:group name = "Bodygroup">
  <xsd:choice>
   <xsd:element ref = "directive.page"/>
   <xsd:element ref = "directive.include"/>
   <xsd:element ref = "scriptlet"/>
   <xsd:element ref = "declaration"/>
   <xsd:element ref = "expression"/>
   <xsd:element ref = "useBean"/>
   <xsd:element ref = "setProperty"/>
   <xsd:element ref = "getProperty"/>
   <xsd:element ref = "include"/>
   <xsd:element ref = "forward"/>
   <xsd:element ref = "plugin"/>
   <xsd:element ref = "text"/>
   <xsd:any namespace="##other" processContents = "lax"/>
  </xsd:choice>
 </xsd:group>


 <!-- jsp:id attribute -->

 <xsd:attribute name = "id" type = "xsd:string"/>

<!--
There should be a constraint for jsp:id to be unique within all elements
```

```
in the document.
  -->

 <!-- Simple types are next -->

 <xsd:simpleType name = "RTE">
   <xsd:annotation>
     <xsd:documentation>
       A request-time expression value
     </xsd:documentation>
   </xsd:annotation>
   <xsd:restriction base = "xsd:string">
     <xsd:pattern value = "&RTexpr;"/>
   </xsd:restriction>
 </xsd:simpleType>

 <xsd:simpleType name = "Bool">
   <xsd:annotation>
     <xsd:documentation>
       Bool would be boolean except it does not accept 1 and 0.
     </xsd:documentation>
   </xsd:annotation>
   <xsd:restriction base = "xsd:NMTOKEN" >
     <xsd:enumeration value = "true"/>
     <xsd:enumeration value = "false"/>
     <xsd:enumeration value = "yes"/>
     <xsd:enumeration value = "no"/>
   </xsd:restriction>
 </xsd:simpleType>

 <xsd:simpleType name = "Identifier">
   <xsd:annotation>
     <xsd:documentation>
       Identifier is an unqualified Java identifier.
     </xsd:documentation>
   </xsd:annotation>
   <xsd:restriction base = "xsd:string">
     <xsd:pattern value = "&Identifier;"/>
   </xsd:restriction>
 </xsd:simpleType>

 <xsd:simpleType name = "TypeName">
   <xsd:annotation>
     <xsd:documentation>
       TypeName is one or more Java identifiers separated by dots
       with no whitespace.
     </xsd:documentation>
   </xsd:annotation>
  <xsd:restriction base = "xsd:string">
    <xsd:pattern value = "&TypeName;"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name = "ImportList">
  <xsd:annotation>
    <xsd:documentation>
      ImportList is one or more typeNames separated by commas.
      Whitespace is allowed before and after the comma.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base = "xsd:string">
    <xsd:pattern value = "&ImportList;"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name = "SetProp">
  <xsd:annotation>
    <xsd:documentation>
      SetProp is an Identifier or *.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base = "xsd:string">
    <xsd:pattern value = "&SetProp;"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name = "RelativeURL">
  <xsd:annotation>
    <xsd:documentation>
      RelativeURL is a uriReference with no colon character
      before the first /, ? or #, if any (RFC2396).
    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base = "xsd:anyURI">
    <xsd:pattern value = "&RelativeURL;"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name = "RTERelativeURL">
  <xsd:union memberTypes = "RelativeURL RTE"/>
</xsd:simpleType>

<xsd:simpleType name = "Length">
  <xsd:annotation>
    <xsd:documentation>
      Length is nn or nn%.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base = "xsd:string">
    <xsd:pattern value = "&Length;"/>
  </xsd:restriction>
</xsd:simpleType>


<xsd:simpleType name = "ExplicitBufferSize">
  <xsd:annotation>
    <xsd:documentation>
       Buffer Size with an explicit value
    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base = "xsd:string">
    <xsd:pattern value = "&Buffer;"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name = "NoneBufferSize">
  <xsd:annotation>
    <xsd:documentation>
       Buffer Size with a "none" value
    </xsd:documentation>
  </xsd:annotation>
     <xsd:restriction base = "xsd:string">
       <xsd:enumeration value = "none"/>
     </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name = "BufferSize">
  <xsd:annotation>
    <xsd:documentation>
      Buffer size is xkb or none.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:union memberTypes = "ExplicitBufferSize NoneBufferSize"/>
</xsd:simpleType>

<xsd:simpleType name = "ContentType">
  <xsd:annotation>
    <xsd:documentation>
      Contetn Type for this page
    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base = "xsd:string">
    <xsd:pattern value = "&ValidContentType;"/>
  </xsd:restriction>
```

```
</xsd:simpleType>

<xsd:simpleType name = "PageEncoding">
  <xsd:annotation>
    <xsd:documentation>
      Page Encoding for this page.  Default is that of ContentType.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base = "xsd:string">
    <xsd:pattern value = "&ValidPageEncoding;"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name = "Scope">
  <xsd:annotation>
    <xsd:documentation>
      valid scope values
    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base = "xsd:NMTOKEN">
    <xsd:enumeration value = "page"/>
    <xsd:enumeration value = "session"/>
    <xsd:enumeration value = "request"/>
    <xsd:enumeration value = "application"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name = "PlugInType">
  <xsd:annotation>
    <xsd:documentation>
      valid values for a plugin type
    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base = "xsd:NMTOKEN">
    <xsd:enumeration value = "bean"/>
    <xsd:enumeration value = "applet"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name = "AlignType">
  <xsd:annotation>
    <xsd:documentation>
      Buffer size is xkb.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base = "xsd:NMTOKEN">
    <xsd:enumeration value = "top"/>
    <xsd:enumeration value = "middle"/>
    <xsd:enumeration value = "bottom"/>
    <xsd:enumeration value = "left"/>
    <xsd:enumeration value = "right"/>
  </xsd:restriction>
</xsd:simpleType>

<!-- Elements follow -->

<xsd:element name = "root">
  <xsd:annotation>
    <xsd:documentation>
      The root element of all JSP documents is named root.

      Authors may, if they wish, include schema location information.
      If specified, the information may appear as attributes of
      the root element as follows:

      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/JSP/Page xsd-file-location"

      Documents should not specify the system identifier of a DTD
      in a DOCTYPE declaration.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:extension base = "Body">
        <xsd:attribute name = "version" fixed = "1.2" type = "xsd:string"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>

<xsd:element name = "directive.page">
  <xsd:annotation>
    <xsd:documentation>
      directive.page is the "page directive".
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:attribute ref = "jsp:id"/>
    <xsd:attribute name = "language" default = "java" type = "xsd:string"/>
    <xsd:attribute name = "extends" type = "TypeName"/>
    <xsd:attribute name = "contentType"
                        default = "text/html; ISO-8859-1" type = "ContentType"/>
    <xsd:attribute name = "pageEncoding"
                        use = "optional" type = "PageEncoding"/>
    <xsd:attribute name = "import" type = "ImportList"/>
    <xsd:attribute name = "session" default = "true" type = "Bool"/>
    <xsd:attribute name = "buffer" default = "8kb" type = "BufferSize"/>
    <xsd:attribute name = "autoFlush" default = "true" type = "Bool"/>
    <xsd:attribute name = "isThreadSafe" default = "true" type = "Bool"/>
    <xsd:attribute name = "info" type = "xsd:string"/>
    <xsd:attribute name = "errorPage" type = "RelativeURL"/>
    <xsd:attribute name = "isErrorPage" default = "false" type = "Bool"/>
  </xsd:complexType>
</xsd:element>

<xsd:element name = "directive.include">
  <xsd:annotation>
    <xsd:documentation>
      directive.include is the "include directive".
      This element does not appear on XML views of JSP pages.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:attribute ref = "jsp:id"/>
    <xsd:attribute name = "file" use = "required" type = "RelativeURL"/>
  </xsd:complexType>
</xsd:element>

<xsd:element name = "scriptlet" type = "BasicType">
  <xsd:annotation>
    <xsd:documentation>
      The representation of a scriplet.
    </xsd:documentation>
  </xsd:annotation>
</xsd:element>

<xsd:element name = "declaration" type = "BasicType">
  <xsd:annotation>
    <xsd:documentation>
      The reprsentation of a declaration.
    </xsd:documentation>
  </xsd:annotation>
</xsd:element>

<xsd:element name = "expression" type = "BasicType">
  <xsd:annotation>
    <xsd:documentation>
      The representation of an expression.
    </xsd:documentation>
  </xsd:annotation>
</xsd:element>

<xsd:element name = "text" type = "BasicType">
```

```
  <xsd:annotation>
    <xsd:documentation>
      Verbatim template text.
    </xsd:documentation>
  </xsd:annotation>
</xsd:element>

<xsd:element name = "useBean">
  <xsd:annotation>
    <xsd:documentation>
      useBean instantiates or accesses a bean in the specified scope.

      Constraint: The allowed combinations of attributes are:

        class [type] | type [( class | beanName)]

    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:extension base="Body">
        <xsd:attribute ref = "jsp:id"/>
        <xsd:attribute name = "id" use = "required" type = "Identifier"/>
        <xsd:attribute name = "class" type = "TypeName"/>
        <xsd:attribute name = "type" type = "TypeName"/>
        <xsd:attribute name = "beanName" type = "TypeName"/>
        <xsd:attribute name = "scope" default = "page" type = "Scope"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>

<xsd:element name = "setProperty">
  <xsd:annotation>
    <xsd:documentation>
      setProperty changes the value of an object property.

      Constraint: 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.

      Exact valid combinations are not expressable in XML Schema.
      They are:

      name="Identifier" property="*"
      name="Identifier" property="Identfiier" param="string"
      name="Identifier" property="Identifier" value="string"
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:attribute ref = "jsp:id"/>
    <xsd:attribute name = "name" use = "required" type = "Identifier"/>
    <xsd:attribute name = "property" use = "required" type = "SetProp"/>
    <xsd:attribute name = "param" type = "xsd:string"/>
    <xsd:attribute name = "value" type = "xsd:string"/>
  </xsd:complexType>
</xsd:element>

<xsd:element name = "getProperty">
  <xsd:annotation>
    <xsd:documentation>
      getProperty obtains the value of an object property.

      Constraint: 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.

    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:attribute ref = "jsp:id"/>
    <xsd:attribute name = "name" use = "required" type = "Identifier"/>
    <xsd:attribute name = "property" use = "required" type = "Identifier"/>
  </xsd:complexType>
</xsd:element>

<xsd:element name = "include">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref = "param" minOccurs = "0" maxOccurs = "unbounded"/>
    </xsd:sequence>
    <xsd:attribute ref = "jsp:id"/>
    <xsd:attribute name = "flush" default = "false" type = "Bool"/>
    <xsd:attribute name = "page" use = "required" type = "RTERelativeURL"/>
  </xsd:complexType>
</xsd:element>

<xsd:element name = "forward">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref = "param" minOccurs = "0" maxOccurs = "unbounded"/>
    </xsd:sequence>
    <xsd:attribute ref = "jsp:id"/>
    <xsd:attribute name = "page" use = "required" type = "RTERelativeURL"/>
   </xsd:complexType>
</xsd:element>

<xsd:element name = "plugin">
  <xsd:complexType> <!-- content only! -->
    <xsd:sequence>
      <xsd:element ref = "params" minOccurs = "0" maxOccurs = "1"/>
      <xsd:element name = "fallback"
                        minOccurs = "0" maxOccurs = "1" type = "Body"/>
    </xsd:sequence>
    <xsd:attribute ref = "jsp:id"/>
    <xsd:attribute name = "type" use = "required" type = "PlugInType"/>
    <xsd:attribute name = "code" type = "xsd:anyURI"/>
    <xsd:attribute name = "codebase" type = "xsd:anyURI"/>
    <xsd:attribute name = "align" type = "AlignType"/>
    <xsd:attribute name = "archive">
      <xsd:simpleType>
        <xsd:list itemType="xsd:anyURI"/>
      </xsd:simpleType>
    </xsd:attribute>
    <xsd:attribute name = "height" type = "Length"/>
    <xsd:attribute name = "hspace" type = "xsd:int"/>
    <xsd:attribute name = "jreversion" default = "1.2" type = "xsd:string"/>
    <xsd:attribute name = "name" type = "xsd:NMTOKEN"/>
    <xsd:attribute name = "vspace" type = "xsd:int"/>
    <xsd:attribute name = "width" type = "Length"/>
    <xsd:attribute name = "nspluginurl" type = "xsd:anyURI"/>
    <xsd:attribute name = "iepluginurl" type = "xsd:anyURI"/>
  </xsd:complexType>
</xsd:element>

<xsd:element name = "params">
  <xsd:complexType>
     <xsd:sequence>
       <xsd:element ref = "param" minOccurs = "1" maxOccurs = "unbounded"/>
     </xsd:sequence>
    <xsd:attribute ref = "jsp:id"/>
  </xsd:complexType>
</xsd:element>

<xsd:element name = "param">
  <xsd:complexType>
    <xsd:attribute ref = "jsp:id"/>
    <xsd:attribute name = "name" use = "required" type = "xsd:NMTOKEN"/>
    <xsd:attribute name = "value" use = "required" type = "xsd:string"/>
  </xsd:complexType>
</xsd:element>
```

```
</xsd:schema>
```

APPENDIX JSP.C

DTD for TagLibrary Descriptor, JSP 1.2

This appendix includes the DTD for a tag library descriptor using JSP 1.2. All JSP 1.2 containers are required to accept such a TLD.

This is the same DTD as "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd", except for some formatting changes to extract comments and make them more readable.

C.1 DTD for TagLibrary Descriptor Files

The following is a DTD describing a Tag Library Descriptor file in JSP 1.2 format.

```
<!--
This is the DTD defining the JavaServer Pages 1.2 Tag Library descriptor (.tld)
(XML) file format/syntax.
A Tag Library is a JAR file containing a valid instance of a Tag Library Descriptor
(taglib.tld) file in the META-INF subdirectory, along with the appropriate imple-
mentation classes and other resources required to implement the actions defined
therein.
Use is subject to license terms.
-->

<!NOTATION WEB-JSPTAGLIB.1_2 PUBLIC “-//Sun Microsystems, Inc.//DTD
JSP Tag Library 1.2//EN”>
```

```
<!--
All JSP 1.2 tag library descriptors must include a DOCTYPE of the following form:
  <!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library
1.2//EN" "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
-->

<!--
The taglib element is the document root, it defines:
```

tlib-version	the version of the tag library implementation
jsp-version	the version of JSP the tag library depends upon
short-name	a simple default name that could be used by a JSP authoring tool to create names with a mnemonicvalue; for example, the it may be used as the preferedprefix 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 small-icon that can be used by tools
large-icon	optional large-icon that can be used by tools
description	a simple string describing the “use” of this taglib, should be user discernable
validator	optional TagLibraryValidator information
listener	optional event listener specification

```
-->

<!ELEMENT taglib (tlib-version, jsp-version, short-name, uri?, display-name?,
small-icon?, large-icon?, description?, validator?, listener*, tag+) >

<!ATTLIST taglib
     id ID #IMPLIED
     xmlns CDATA #FIXED “http://java.sun.com/JSP/TagLibraryDescriptor”>

<!--
The value of the tlib-version element describes this version (number) of the tagl-
ibrary. This element is mandatory.
```

```
#PCDATA ::= [0-9]*{ “.”[0-9] }0..3
-->

<!ELEMENT tlib-version (#PCDATA)

<!--
The value of the jsp-version element describes the JSP version (number) this
taglibrary requires in order to function. This element is mandatory. The value that
should be used for JSP 1.2 is "1.2" (no quotes).

#PCDATA ::= [0-9]*{ “.”[0-9] }0..3
-->

<!ELEMENT jsp-version (#PCDATA) >

<!--
The value of the short-name element is a name that could be used by a JSP au-
thoring tool to create names with a mnemonic value; for example, it may be used
as the prefered prefix value in taglib directives.
Do not use white space, and do not start with digits or underscore.

#PCDATA ::= NMTOKEN
-->

<!ELEMENT short-name (#PCDATA) >

<!--
The value of the uri element is a public URI that uniquely identifies the exact se-
mantics of this taglibrary.
-->

<!ELEMENT uri (#PCDATA) >

<!--
The value of the description element is an arbitrary text string describing the tag
library.
-->

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

```
<!--
The validator element provides information on an optional validator that can be
used to validate the conformance of a JSP page to using this tag library.
-->

<!ELEMENT validator (validator-class, init-param*, description?) >

<!--
The validator-class element defines the TagLibraryValidator class that can be
used to validate the conformance of a JSP page to using this tag library.
-->

<!ELEMENT validator-class (#PCDATA) >

<!--
The init-param element contains a name/value pair as an
initialization param.
-->

<!ELEMENT init-param (param-name, param-value, description?)>

<!--
The param-name element contains the name of a parameter.
-->

<!ELEMENT param-name (#PCDATA)>

<!--
The param-value element contains the value of a parameter.
-->

<!ELEMENT param-value (#PCDATA)>

<!--
The listener element defines an optional event listener object to be instantiated
and
registered automatically.
-->

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

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

<!ELEMENT listener-class (#PCDATA) >

<!--
The *tag* element defines an action in this tag library. The tag element has one attribute, id.
The tag element may have several subelements defining:

name	The unique action name
tag-class	The tag handler class implementing javax.servlet.jsp.tagext.Tag
tei-class	An optional subclass of javax.servlet.jsp.tagext.TagExtraInfo
body-content	The body content type
display-name	A short name that is intended to be displayed by tools
small-icon	Optional small-icon that can be used by tools
large-icon	Optional large-icon that can be used by tools
description	Optional tag-specific information
variable	Optional scripting variable information
attribute	All attributes of this action
example	Optional informal description of an example of a use of this action.

-->

<!ELEMENT tag (name, tag-class, tei-class?, body-content?, display-name?, small-icon?, large-icon?, description?, variable*, attribute*, example?) >

<!--
The *tag-class* element indicates the subclass of javax.serlvet.jsp.tagext.Tag that implements the request time semantics for this tag. This element is required.

#PCDATA ::= fully qualified Java class name
-->

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

<!--
The *tei-class* element indicates the subclass of javax.servlet.jsp.tagext.TagExtraInfo for this tag. The class is instantiated at translation time. This element is optional.

#PCDATA ::= fully qualified Java class name
-->

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

<!--
The *body-content* element provides provides information on the content of the body of this tag. This element is primarily intended for use by page composition tools.
There are currently three values specified:

tagdependent	The body of the tag is interpreted by the tag implementation itself, and is most likely in a different “langage”, e.g embedded SQL statements.
JSP	The body of the tag contains nested JSP syntax
empty	The body must be empty

This element is optional; the default value is JSP

#PCDATA ::= tagdependent | JSP | empty
-->

```
<!ELEMENT body-content (#PCDATA) >
```

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

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

<!--
The *large-icon* element contains the name of a file containing a large (32 x 32) icon image. The icon can be used by tools. The file name is a relative path within the tag library.
The image must be either in the JPEG or GIF format, and the file name must end with the suffix “.jpg” or “.gif” respectively.
-->

<!ELEMENT large-icon (#PCDATA) >

<!--
The *small-icon* element contains the name of a file containing a small (16 x 16) icon image. The icon can be used by tools. The file name is a relative path within the tag library.
The image must be either in the JPEG or GIF format, and the file name must end with the suffix “.jpg” or “.gif” respectively.
-->

<!ELEMENT small-icon (#PCDATA) >

<!--
The *example* element provides an informal description of an example of the use of a tag.
-->

<!ELEMENT example (#PCDATA) >

<!--
The *variable* element provides information on the scripting variables defined by this tag.
It is a (translation time) error for an action that has one or more variable 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.
variable-class	Name of the class of the variable. java.lang.String is default.

declare	Whether the variable is declared or not. True is the default.
scope	The scope of the scripting variable defined. NESTED is default.

-->

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

<!--
The *name-given* element provides the name for the scripting variable.
One of name-given or name-from-attribute is required.
-->

<!ELEMENT name-given (#PCDATA) >

<!--
The value of the *name-from-attribute* element is the name of an attribute whose (translation-time) value will give the name of the variable.
One of name-given or name-from-attribute is required.
-->

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

<!--
The *variable-class* element is the name of the class for the scripting variable.
This element is optional; the default is java.lang.String.
-->

<!ELEMENT variable-class (#PCDATA) >

<!--
The value of the *declare* element indicates whether the scripting variable is to be defined or not. See TagExtraInfo for details.
This element is optional and is the default is true.
-->

<!ELEMENT declare (#PCDATA) >

<!--
The value of the *scope* element describes the scope of the scripting variable.

```
See TagExtraInfo for details.
This element is optional and the default value is the string “NESTED”. The other
legal values are “AT_BEGIN” and “AT_END”.
-->

<!ELEMENT scope (#PCDATA) >

<!--
The attribute element defines an attribute for the nesting tag.
The attributre element may have several subelements defining:

name          the name of the attribute

attribute     whether the attribute is required or optional

rtexpravaluewhether the attribute is a runtime attribute

type          the type of the attributes

description   a description of the attribute

-->

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

<!--
The name element defines the canonical name of a tag or attribute being defined

#PCDATA ::= NMTOKEN
-->

<!ELEMENT name(#PCDATA) >

<!--
The value of the required element indicates if the nesting attribute is required or
optional. This attribute is optional and its default value is false.

#PCDATA ::= true | false | yes | no
-->

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

```
<!--
The value of the rtexpvalue element indicates if the value of the attribute may be
dynamically calculated at request time, as opposed to a static value determined
at translation time. This attribute is optional and its default value is false

#PCDATA ::= true | false | yes | no
-->

<!ELEMENT rtexprvalue (#PCDATA) >

<!--
The value of the type element describes the Java type of the attributes value.
For static values (those determined at translation time) the type is always ja-
va.lang.String.
-->

<!ELEMENT type (#PCDATA) >

<!-- ID attributes -->

<!ATTLIST tlib-version id ID #IMPLIED>
<!ATTLIST jsp-version id ID #IMPLIED>
<!ATTLIST short-name id ID #IMPLIED>
<!ATTLIST uri id ID #IMPLIED>
<!ATTLIST description id ID #IMPLIED>
<!ATTLIST example id ID #IMPLIED>
<!ATTLIST tag id ID #IMPLIED>
<!ATTLIST tag-class id ID #IMPLIED>
<!ATTLIST tei-class id ID #IMPLIED>
<!ATTLIST body-content id ID #IMPLIED>
<!ATTLIST attribute id ID #IMPLIED>
<!ATTLIST name id ID #IMPLIED>
<!ATTLIST required id ID #IMPLIED>
<!ATTLIST rtexprvalue id ID #IMPLIED>
<!ATTLIST param-name id ID #IMPLIED>
<!ATTLIST param-value id ID #IMPLIED>
<!ATTLIST listener id ID #IMPLIED>
<!ATTLIST listener-class id ID #IMPLIED>
```

A P P E N D I X JSP.D

DTD for TagLibrary Descriptor, JSP 1.1

This appendix includes the DTD for a tag library descriptor using JSP 1.1. All JSP 1.2 containers are required to accept such a TLD.

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

D.1 DTD for TagLibrary Descriptor Files

The following is a DTD describing a Tag Library Descriptor file in JSP 1.1 format.

```
<!--
This is the DTD defining the JavaServer Pages 1.1 Tag Library descriptor (.tld)
(XML) file format/syntax.

A Tag Library is a JAR file containing a valid instance of a Tag Library Descriptor
(taglib.tld) file in the META-INF subdirectory, along with the appropriate imple-
menting classes, and other resources required toimplement the tags defined
therein.

Use is subject to license terms.
  -->

<!--
The taglib tag is the document root, it defines:
```

```
tlibversion    the version of the tag library implementation
jspversion     the version of JSP the tag library depends upon
shortname      a simple default short name that could be used by a JSP authoring
        tool to create names with a mnemonic value; for example, the it may be used
        as the prefered prefix value in taglib directives
uri            a uri uniquely identifying this taglib
info           a simple string describing the “use” of this taglib, should be user
        discernable
-->

<!ELEMENT taglib (tlibversion, jspversion?, shortname, uri?, info?, tag+) >
<!ATTLIST taglib id ID #IMPLIED
         xmlns CDATA #FIXED
              “http://java.sun.com/dtd/web-jsptaglibrary_1_1.dtd”
>

<!--
Describes this version (number) of the taglibrary (dewey decimal)
#PCDATA ::= [0-9]*{ “.”[0-9] }0..3
-->

<!ELEMENT tlibversion (#PCDATA) >

<!--
Describes the JSP version (number) this taglibrary requires in order to function
(dewey decimal)
The default is 1.1

#PCDATA ::= [0-9]*{ “.”[0-9] }0..3
-->

<!ELEMENT jspversion  (#PCDATA) >

<!--
Defines a short (default) shortname to be used for tags and variable names used/
created by this tag library.  Do not use white space, and do not start with digits or
underscore.

#PCDATA ::= NMTOKEN
-->
```

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

<!--
Defines a public URI that uniquely identifies this version of the taglibrary Leave it
empty if it does not apply.
-->

<!ELEMENT uri (#PCDATA) >

<!--
Defines an arbitrary text string descirbing the tag library
-->

<!ELEMENT info(#PCDATA) >

<!--
The tag defines a unique tag in this tag library, defining:
- the unique tag/element name
- the subclass of javax.servlet.jsp.tagext.Tag implementation class
- an optional subclass of javax.servlet.jsp.tagext.TagExtraInfo
- the body content type (hint)
- optional tag-specific information
- any attributes
-->

<!ELEMENT tag (name, tagclass, teiclass?, bodycontent?, info?, attribute*) >

<!--
Defines the subclass of javax.serlvet.jsp.tagext.Tag that implements the request
time semantics for this tag. (required)

#PCDATA ::= fully qualified Java class name
-->

<!ELEMENT tagclass (#PCDATA) >

<!--
Defines the subclass of javax.servlet.jsp.tagext.TagExtraInfo for this tag. (option-
al)
If this is not given, the class is not consulted at translation time.
```

```
#PCDATA ::= fully qualified Java class name
-->

<!ELEMENT teiclass (#PCDATA) >

<!--
Provides a hint as to the content of the body of this tag. Primarily intended for use
by page composition tools.
There are currently three values specified:
tagdependent     The body of the tag is interpreted by the tag implementation
      itself, and is most likely in a different “langage”, e.g embedded SQL
      statements.
JSP              The body of the tag contains nested JSP syntax
empty            The body must be empty. The default (if not defined) is JSP

#PCDATA ::=  tagdependent | JSP | empty
-->

<!ELEMENT bodycontent (#PCDATA) >

<!--
The attribute tag defines an attribute for the nesting tag
An attribute definition is composed of:
- the attributes name (required)
- if the attribute is required or optional (optional)
- if the attributes value may be dynamically calculated at runtime
      by a scriptlet expression (optional)
-->

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

<!--
Defines the canonical name of a tag or attribute being defined

#PCDATA ::= NMTOKEN
-->

<!ELEMENT name(#PCDATA) >

<!--
Defines if the nesting attribute is required or optional.
```

```
#PCDATA ::= true | false | yes | no

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

<!ELEMENT required    (#PCDATA) >

<!--
Defines if the nesting attribute can have scriptlet expressions as a value, i.e the
value of the attribute may be dynamically calculated at request time, as opposed
to a static value determined at translation time.

#PCDATA ::= true | false | yes | no

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

<!ELEMENT rtexprvalue (#PCDATA) >

<!ATTLIST tlibversion id ID #IMPLIED>
<!ATTLIST jspversion id ID #IMPLIED>
<!ATTLIST shortname id ID #IMPLIED>
<!ATTLIST uri id ID #IMPLIED>
<!ATTLIST info id ID #IMPLIED>
<!ATTLIST tag id ID #IMPLIED>
<!ATTLIST tagclass id ID #IMPLIED>
<!ATTLIST teiclass id ID #IMPLIED>
<!ATTLIST bodycontent id ID #IMPLIED>
<!ATTLIST attribute id ID #IMPLIED>
<!ATTLIST name id ID #IMPLIED>
<!ATTLIST required id ID #IMPLIED>
<!ATTLIST rtexprvalue id ID #IMPLIED>
```

APPENDIX JSP.E
Changes

This appendix lists the changes in the JavaServer Pages specification. This appendix is non-normative.

E.1 Changes Between PFD 2 and Final Draft

This is the final version approved by JCP Executive Comittee; the document was updated to reflect that status. All change bars were reset.

E.1.1 Added jsp:id mechanism

A new mechanism was added to allow willing JSP containers to provide improved translation-time error information from TagLibraryValidator classes. The signature of TagLibraryValidator.validate() was modified slightly, and a new ValidationMessage class was added. These objects act through a new attribute, jsp:id, which is optionally supported by a JSP container and exposed only through the XML view of a JSP page. Chapter JSP.5 (mostly Section JSP.5.3.13), Chapter JSP.7 (Section JSP.7.5.1.2) and Chapter JSP.10 (Section JSP.10.5.5) were affected.

E.1.2 Other Small Changes

- Made height & width be rtexprs. Section JSP.4.7 was affected.
- Added attribute value conversion from String literal to short and Short, and corrected conversion for char and Character in Table JSP.2-2.
- Corrected a statement on the allowed return values for doStartTag() for Tag, IterationTag and BodyTag.. PFD2 incorrectly indicated that "emtpy" tags

could only return SKIP_BODY; the correct statement is that tags whose body-content is "empty" can only return SKIP_BODY.

E.1.3 Clarification of role of id

The mandated interpretations of the "id" attribute in Section JSP 2.13.3 (that id represents page-wide unique ids) and the "scope" attribute in Section JSP 2.13.4 (regarding the scope for the introduced variable) were not enforced by most (perhaps all?) containers, and were inconsistent with prevalent practices in custom tag library development. Essentially these sections were being interpreted as localized statements about the jsp:useBean standard action. This has been made explicit and the sections were moved to Chapter 4 to reflect that.

Sections JSP.2.13.3 and JSP.2.13.4, and Chapter 4 were affected.

E.1.4 Clarifications on Multiple Requests and Threading

- Clarify that TLV instances need be thread safe. This affected Section JSP.10.5.5.
- Clarify that a tag handler instance is actively processing only one request at a time; this happens naturally if the tag handler is instantiated afresh through new() invocations, but it requires spelling once tag handler pooling is introduced. This clarification affected Section JSP.10.1.

E.1.5 Clarifications on JSP Documents

Several clarifications in Chapter 5.

- Reafirmed that, in a JSP page in XML syntax, the URI for jsp core actions is important, not the prefix.
- Clarify that <?xml ... ?> is not required (as indicated by the XML spec).
- Clarified further the interpretation of whitespace on JSP documents.

E.1.6 Clarifications on Well Know Tag Libraries

Clarified that a tag library author may indicate, through the description comment, that a tag handler may expose at runtime only some subset of the information described through the tag handler implementation class. This is useful for specialized implementations of well-known tag libraries like the JSP standard tag library. This clarification affected the description of the tag element in

Section JSP.7.4 and the description of Tag.setParent() and TagSupport.findAncestorWithClass().

Removed the last paragraph on Section JSP.7.3.9; we don't have any plans to remove the well-know URI mechanism.

In general cleaned up the presentation of the computation of the taglib map between a URI and a TLD resource path; the previous version was clunky.

E.1.7 Clarified Impact of Blocks

Clarified further the legal uses and the role of block constructs within scriptlets and nested actions. This affected small portions of Sections JSP.2.3.3, JSP.6.4, JSP.6.4.4 and JSP.10.5.9.

E.1.8 Other Small Clarifications

- Reafirmed more explicitly that the location of icons is relative to TLD file. Section JSP.7.4 was affected.
- Removed non-normative comment about JSR-045 in Section JSP.2.1.6.
- Removed the comment on errorPages needing to be JSP pages, they can also be static objects. This affects Table JSP.2-1.
- Reaffirmed that event listeners in a tag library are registered before the application is started. This affects Section JSP.7.1.2.2.
- Clarify when the use of quoting conventions is required for attribute values. Clarified that request-time attribute values follow the same rules. This affects Section JSP.2.3.5, Section JSP.2.6 and Section JSP.2.13.1.
- Clarified the interpretation of relative specifications for include directives and jsp:include and jsp:forward actions. This affected Section JSP.2.2.1, Section JSP.2.10.4, Section JSP.4.4 and Section JSP.4.5
- Corrected the inconsistency on the precompilation protocol in Section JSP.8.4.2 regarding whether the requests are delivered to the page or not; they are not.
- Clarified that the <type> subelement of <attribute> in the TLD file should match that of the underlying JavaBean component property.
- Spelled out the use of ClassLoader.getResource() to get at data from a TagLibraryValidator class.

E.2 Changes Between 1.2 PFD 1b and PFD 2

Change bars are used in almost all chapters to indicate changes between PFD 1b and PFD 2. The exception are Chapters 9 and 10 which are generated automatically from the Java sources and have no change bars. Most changes are semantical, but some of them are editorial.

E.2.1 Added elements to Tag Library Descriptor

The Tag Library Descriptor (TLD) was extended with descriptive information that is useful to users of the tag library. In particular, a TLD can now be massaged directly (e.g. using an XSLT stylesheet) into an end-user document.

A new <example> element was added, as an optional subelement of <tag>. The existing <description> element was made a valid optional subelement of <variable>, <attribute> and <validator>.

Section JSP.7.4 and Appendix JSP.C were affected. The TLD 1.2 DTD and Schemas were also affected.

E.2.2 Changed the way version information is encoded into TLD

The mechanism used to provide version information on the TLD was changed. In the PFD the version was encoded into the namespace. In PFD2 the namespace is not intended to change unless there are non-compatible changes, and the version is encoded into the <jsp-version> element, which is now mandatory. The new URI for the namespace is "http://java.sun.com/JSP/TagLibraryDescriptor".

Chapter JSP.7 and Appendix JSP.C were affected.

E.2.3 Assigning String literals to Object attributes

It is now possible to assign string literals to an attribute that is defined as having type Object, as well as to a property of type Object. The valid type conversions are now all described in Section JSP.2.13.2, and used by reference in the semantics of <jsp:setProperty>.

E.2.4 Clarification on valid names for prefix, action and attributes

We clarified the valid names for prefixes used in taglib directives, element names used in actions, and attribute names.

E.2.5 Clarification of details of empty actions

The JSP 1.1 specification distinguishes empty from non-empty actions, although the description could be better. Unfortunately, the JSP 1.2 PFD1 draft did not improve the description. This draft improves the description by making it clear what methods are invoked when.

Chapters 2, 7 and 10 were affected.

E.2.6 Corrections related to XML syntax

We clarified several issues related to the XML syntax for JSP pages and to the XML view of a JSP page. Most changes are in Chapter 5.

- Removed an inexistant flush attribute in the include directive at Section JSP.5.2.4().
- Changed the name of jsp:cdata to jsp:text, since its semantics are very similar to the text element in XSLT.
- Changed the way the version information is encoded into the XML syntax; the URI used now is not version-specific and instead there is a required version attribute of jsp:root.
- Clarified that JSP comments in a JSP page in JSP syntax are not preserved on the XML view of the page.
- Clarified that JSP pages in XML syntax should have no DOCTYPE.
- Clarified the treatment of include directives in the XML view of a JSP page.
- Clarified the format of the URIs to use in xmlns attributes for taglib directives, and corrected Appendix JSP.B.

E.2.7 Other changes

We clarified several other inconsistencies or mistakes

- Explicitly indicated which attributes are reserved (Section JSP.2.3.5) and which prefixes are reserved (Section JSP.2.10.2).
- Add a comment to the DTD for the TLD indicating that a DOCTYPE is needed and what its value is. No changes to the value.
- Removed the paragraph at the end of Section JSP.7.3.9 that used to contain non-normative comments on the future of "well kwown URIs".
- Corrected the description of the valid values that can be passed to the flush attribute of the include action in Section JSP.4.4.
- Clarified that <jsp:param> can only appear within <jsp:forward>, <jsp:include>, and <jsp:params>.
- Clarified that <jsp:params> and <jsp:fallback> can only appear within <jsp:plugin>.
- Resolved a conflict in Section JSP.4.4 between the Servlet and the JSP specification regarding how to treat modifications to headers in included actions.
- Section 10.1.1 in PFD1 incorrectly described the valid return values for doStartTag() in tag handlers that implement the BodyTag interface. The correct valid values are SKIP_BODY, EVAL_BODY_INCLUDE and EVAL_BODY_BUFFER. Section now indicates this.

E.3 Changes Between 1.2 PFD and 1.2 PFD 1b

PFD 1b is a draft that has mostly formating and a few editorial changes. This draft is shown only to make it simpler to correlate changes between later drafts and the previous drafts.

Change bars are used to indicate changes between PFD 1 and PFD 1b.

E.4 Changes Between 1.2 PD1 and 1.2 PFD

The following changes ocurred between the Public Draft 1 and the Proposed Final Draft versions of the JSP 1.2 specification.

E.4.1 Deletions

- Removed the resetCustomAttributes() method.

E.4.2 Additions

- Added constructors and methods to JspException to support a rootCause (paralleling the ServletException).
- Added a PageContext.handleException(Throwable) method.
- Added references to JSR-045 regarding debugging support.
- Added new TryCatchFinally interface to provide better control over exceptions in tag handlers.
- Added an implicit URI to TLD map for packaged tag libraries. This also provides support for multiple TLDs inside a single JAR file.
- Added pageEncoding attribute to page directive.
- Added material to Chapter JSP.3.
- Added TagValidatorInfo class.
- Added Section JSP.2.1.7 with a suggestion on extension convention for top and included JSP files.

E.4.3 Clarifications

- A tag handler object can be created with a simple “new()”; it needs not be a fully fledged Beans, supporting the complete behavior of the java.beans.Beans.instantiate() method.
- Removed the “recommendation” that the <uri> element in a TLD be a URL to anything.
- Clarified that extension dependency information in packaged tag libraries should be honored.
- Clarified invocation and lifecycle of TagLibraryValidator.
- Clarified where TLDs may appear in a packaged JAR file.
- Clarified when are response.getWriter().

E.4.4 Changes

- Moved a couple of chapters around
- Improved and clarified Chapter JSP.5.
- Moved the include directive back into Chapter JSP.2.
- Renamed javax.servlet.jsp.tagext.PageInfo to javax.servlet.jsp.tagext.Page-Data (for consistency with existing TagData).
- Added initialization parameters to TagLibraryInformation validation in TLD, adding a new <validator> element, renaming <validatorclass> to <validator-class> for consistency, and adding <init-param> as in the Servlet web.xml descriptor.
- Added method to pass the initialization parameters to the validator class and removed the use of TagLibraryInfo. Added prefix and uri String arguments to validate() method.
- Changed element names in TLD to consistently follow convention. New names are <tag-class>. <tei-class>, <tlib-version, <jsp-version>, <short-name> and <body-content>. <info> was renamed <description>.

E.5 Changes Between 1.1 and 1.2 PD1

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

E.5.1 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.

E.5.2 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.

E.5.3 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.

E.5.4 Clarifications

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

E.5.5 Changes

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

E.6 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.

E.6.1 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 JSP.F.
- Expanded Chapter 1 so as to cover 0.92’s "model 1" and "model 2".
- Clarified a number of JSP 1.0 details.

E.6.2 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 JSP.F Glossary

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

action An element in a JSP page that can act on implicit objects and other server-side objects or can define new scripting variables. Actions follow the XML syntax for elements with a start tag, a body and an end tag; if the body is empty it can also use the empty tag syntax. The tag must use a prefix.

action, standard An action that is defined in the JSP specification and is always available to a JSP file without being imported.

action, custom An action described in a portable manner by a tag library descriptor and a collection of Java classes and imported into a JSP page by a taglib directive.

Application Assembler A person that combines JSP pages, servlet classes, HTML content, tag libraries, and other Web content into a deployable Web application.

component contract The contract between a component and its container, including life cycle management of the component and the APIs and protocols that the container must support.

Component Provider A vendor that provides a component either as Java classes or as JSP page source.

distributed container A JSP container that can run a Web application that is tagged as distributable and is spread across multiple Java virtual machines that might be running on different hosts.

declaration A scripting element that declares methods, variables, or both in a JSP page. Syntactically it is delimited by the <%! and %> characters.

directive An element in a JSP page that gives an instruction to the JSP container and is interpreted at translation time. Syntactically it is delimited by the <%@ and %> characters.

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, frontA JSP page that receives an HTTP request directly from the client. It creates, updates, and/or accesses some server-side data and then forwards the request to a presentation JSP page.

JSP page, presentation A JSP page that is intended for presentation purposes only. It accesses and/or updates some server-side data and incorporates fixed template data to create content that is sent to the client.

JSP page implementation class The Java programming language class, a Servlet, that is the runtime representation of a JSP page and which receives the *request* object and updates the *response* object. The page implementation class can use the services provided by the JSP container, including both the Servlet and the JSP APIs.

JSP page implementation object The instance of the JSP page implementation class that receives the request object and updates the response object.

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.