JAKARTA EE

Jakarta Server Pages

Jakarta Server Pages Team, https://projects.eclipse.org/projects/ee4j.jsp

3.0-RC2, June 13, 2020: DRAFT

Table of Contents

```
Specification: Jakarta Server Pages

Version: 3.0-RC2

Status: DRAFT

Release: June 13, 2020
```


Eclipse Foundation Specification License

By using and/or copying this document, or the Eclipse Foundation document from which this statement is linked, you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions:

Permission to copy, and distribute the contents of this document, or the Eclipse Foundation document from which this statement is linked, in any medium for any purpose and without fee or royalty is hereby granted, provided that you include the following on ALL copies of the document, or portions thereof, that you use:

- link or URL to the original Eclipse Foundation document.
- All existing copyright notices, or if one does not exist, a notice (hypertext is preferred, but a textual representation is permitted) of the form: "Copyright © [$date-of-document] Eclipse Foundation, Inc. <<url to this license>>"

Inclusion of the full text of this NOTICE must be provided. We request that authorship attribution be provided in any software, documents, or other items or products that you create pursuant to the implementation of the contents of this document, or any portion thereof.

No right to create modifications or derivatives of Eclipse Foundation documents is granted pursuant to this license, except anyone may prepare and distribute derivative works and portions of this document in software that implements the specification, in supporting materials accompanying such software, and in documentation of such software, PROVIDED that all such works include the notice below. HOWEVER, the publication of derivative works of this document for use as a technical specification is expressly prohibited.

The notice is:

"Copyright © 2018, 2020 Eclipse Foundation. This software or document includes material copied from or derived from Jakarta ® Server Pages https://jakarta.ee/specifications/pages/3.0/ [https://jakarta.ee/specifications/pages/3.0/]"

Disclaimers

THIS DOCUMENT IS PROVIDED "AS IS," AND THE COPYRIGHT HOLDERS AND THE ECLIPSE FOUNDATION MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DOCUMENT ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

THE COPYRIGHT HOLDERS AND THE ECLIPSE FOUNDATION WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE DOCUMENT OR THE PERFORMANCE OR IMPLEMENTATION OF THE CONTENTS THEREOF.

The name and trademarks of the copyright holders or the Eclipse Foundation may NOT be used in advertising or publicity pertaining to this document or its contents without specific, written prior permission. Title to copyright in this document will at all times remain with copyright holders.

Jakarta Server Pages Specification, Version 3.0

The Jakarta Server Pages Team - June 13, 2020

Comments to: jsp-dev@eclipse.org [mailto:jsp-dev@eclipse.org]

Preface

This is the Jakarta Server Pages specification version 3.0, developed by the Jakarta Server Pages Team under the Eclipse Foundation Specification Process.

Who Should Read This Document

This document is the authoritative JSP 3.0 specification. It is intended to provide requirements for implementations of JSP page processing, and support by web containers in web servers and application servers. As an authoritative document, it covers material pertaining to a wide audience, including *Page Authors*, *Tag Library Developers*, *Deployers*, *Container Vendors*, and *Tool Vendors*.

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

Organization of This Document

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

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

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

Finally, Part III contains all the appendices.

Historical Note

Prior to version 3.0, this specification was developed under the Java Community Process as part of JSR 245.

Overview

The Jakarta Server Pages Technology

Jakarta Server Pages (JSP) is the Jakarta EE technology for building applications for generating dynamic web content, such as HTML, DHTML, XHTML, and XML. JSP technology enables the easy authoring of web pages that create dynamic content with maximum power and flexibility.

General Concepts

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

- *Template Data*
  A substantial portion of most dynamic content is fixed or *template* content. Text or XML fragments are typical template data. JSP technology supports natural manipulation of template data.
- *Addition of Dynamic Data*
  JSP technology provides a simple, yet powerful, way to add dynamic data to template data.
- *Encapsulation of Functionality*
  JSP technology provides two related mechanisms for the encapsulation of functionality: JavaBeans™ component architecture, and tag libraries delivering custom actions, functions, listener classes, and validation.
- *Good Tool Support*
  Good tool support leads to significantly improved productivity. Accordingly, JSP technology has features that enable the creation of good authoring tools.

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

Benefits of Jakarta Server Pages Technology

JSP technology offers the following benefits:

- *Write Once, Run Anywhere properties*
  JSP technology is platform independent in its dynamic web pages, its web servers, and its underlying server components. JSP pages may be authored on any platform, run on any web server or web enabled application server, and accessed from any web browser. Server components can be built on any platform and run on any server.
- *High quality tool support*
  Platform independence allows the JSP user to choose best-of-breed tools. Additionally, an explicit goal of the Jakarta Server Pages design is to enable the creation of high quality portable tools.
- *Separation of Roles*
  JSP supports the separation of developer and author roles. *Developers* write components that interact with server-side objects. *Authors* put static data and dynamic content together to create presentations suited for their intended audience.

Each group may do their job without knowing the job of the other. Each role emphasizes different abilities and, although these abilities may be present in the same individual, they most commonly will not be. Separation allows a natural division of labor.
A subset of the developer community may be engaged in developing reusable components intended to be used by authors.

- *Reuse of components and tag libraries*
  Jakarta Server 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*
  Jakarta Server Pages technology enables the separation of static content in a template from dynamic content that is inserted into the static template. This greatly simplifies the creation of content. The separation is supported by beans specifically designed for the interaction with server-side objects, and by the tag extension mechanism.
- *Support for actions, expressions, and scripting*
  Jakarta Server Pages technology supports scripting elements as well as actions. Actions encapsulate useful functionality in a convenient form that can be manipulated by tools. Expressions are used to access data. Scripts can be used to glue together this functionality in a per-page manner.
  Expressions in the EL directly express page author concepts like properties in beans and provide more controlled access to the Web Application data. Functions defined through the tag library mechanism can be accessed in the EL.
  Page authors can write actions using the JSP technology directly. This greatly increases the ease with which action abstractions can be created.
- *Web access layer for N-tier enterprise application architecture(s)*
  Jakarta Server Pages technology is an integral part of Jakarta EE. The Jakarta EE platform brings Java technology to enterprise computing. One can now develop powerful middle-tier server applications that include a web site using Jakarta Server Pages technology as a front end to Enterprise JavaBeans components in a Jakarta EE compliant environment.

Basic Concepts

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

What Is a JSP Page?

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

- Standard directives

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

Web Applications

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

- Java Runtime Environment(s) running on the server (required)
- JSP page(s) that handle requests and generate dynamic content
- Servlet(s) that handle requests and generate dynamic content
- Server-side JavaBeans components that encapsulate behavior and state
- Static HTML, DHTML, XHTML, XML, and similar pages.

The Jakarta Server Pages specification inherits from the servlet specification the concepts of web applications, servlet contexts, sessions, and requests and responses. See the Jakarta Servlet 5.0 specification for more details.

Components and Containers

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

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

Translation and Execution Steps

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

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

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

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

Deployment Descriptor and Global Information

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

Role in Jakarta EE

With a few exceptions, integration of JSP pages within Jakarta EE 9 is inherited from the Servlet 5.0 specification since translation turns JSPs into servlets.

Users of Jakarta Server Pages

There are six classes of users that interact with Jakarta Server Pages technology. This section describes each class of user, enumerates the technologies each must be familiar with, and identifies which sections of this specification are most relevant to each user class. The intent is to ensure that Jakarta Server Pages remains a practical and easy-to-use technology for each class of user, even as the language continues to grow.

Page Authors

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

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

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

Advanced Page Authors

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

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

- Chapters Chapter 1, *Core Syntax and Semantics*, Chapter 2, *Expression Language*, Chapter 3, *JSP Configuration*, Chapter 4, *Internationalization Issues* and Chapter 5, *Standard Actions*
- Chapter 6, *JSP Documents*
- Section 9.1.1, “Valid JSP Page” and Section 9.1.2, “Reserved Names” of Chapter 9, *Scripting*
- Section 7.1.1, “Goals” and Section 7.1.2, “Overview” of Chapter 7, *Tag Extensions*
- Chapter 8, *Tag Files*
- Section 11.4, “Precompilation” of Chapter 11, *JSP Container*
- Appendices Appendix A, *Packaging JSP Pages*, Appendix C, *Changes*, and Appendix D, *Glossary*

Advanced page authors may also wish to look at the Javadoc for the `jakarta.servlet.jsp` package and the XML schema for the JSP 3.0 deployment descriptor.

Tag Library Developers

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

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

- Chapters Chapter 1, *Core Syntax and Semantics*, Chapter 2, *Expression Language*, Chapter 3, *JSP Configuration*, Chapter 4, *Internationalization Issues* and Chapter 5, *Standard Actions*
- Chapter 6, *JSP Documents*
- Section 9.1.1, “Valid JSP Page” and Section 9.1.2, “Reserved Names” of Chapter 9, *Scripting*
- Chapter 7, *Tag Extensions*
- Chapter 8, *Tag Files*
- Section 11.4, “Precompilation” of Chapter 11, *JSP Container*
- All Appendices

Tag library developers may also wish to look at the Javadoc for the `jakarta.servlet.jsp` and `jakarta.servlet.jsp.tagext` packages.

Deployers

A deployer is an expert in a specific operational environment who is responsible for configuring a web application for, and deploying the web application to, that environment. The deployer does not need to understand the target language or Java, but must have an understanding of XML or use tools that provide the ability to read deployment descriptors.

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

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

Container Developers and Tool Vendors

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

Part I

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

The chapters of this part are:

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

Chapter 1. Core Syntax and Semantics

This chapter describes the core syntax and semantics for the Jakarta Server Pages 3.0 specification (JSP 3.0).

1.1. What Is a JSP Page

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

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

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

1.1.1. Web Containers and Web Components

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

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

1.1.2. Generating HTML

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

1.1.3. Generating XML

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

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

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

1.1.4. Translation and Execution Phases

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

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

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

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

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 1.1.9, "Compiling JSP Pages" describes how to perform the translation phase ahead of deployment.

1.1.5. Validating JSP pages

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

The structure of the custom actions in a JSP page is always exposed in the XML view. This means that a

tag library validator can check that, for instance, some custom actions are only used within others.

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

1.1.6. Events in JSP Pages

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

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

1.1.7. JSP Configuration Information

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

1.1.8. Naming Conventions for JSP Files

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

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

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

- The `.jsp` extension is used only for files corresponding to top level JSP files, forming a JSP page

when processed.

- Statically included segments use any other extension. As included segments were called 'JSP fragments' in past versions of this specification, the extension `.jspf` was offered as a suggestion. This extension is still suggested for consistency reasons, despite that they are now called 'JSP segments'.

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

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

1.1.9. Compiling JSP Pages

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

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

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

A JSP page can also be compiled at deployment time.

1.1.9.1. JSP Page Packaging

When a JSP page implementation class depends on support classes in addition to the JSP 3.0 and Servlet 5.0 classes, the support classes are included in the packaged WAR, as defined in the Servlet 5.0 specification, for portability across JSP containers.

Appendix A, *Packaging JSP Pages* contains two examples of JSP pages packaged in WARs:

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

1.1.10. Debugging JSP Pages

In the past debugging tools provided by development environments have lacked a standard format for conveying source map information allowing the debugger of one vendor to be used with the JSP

container of another. JSP 3.0 containers must support the Jakarta Debugging Support for Other Languages Specification. Details can be found in Section 11.5, “Debugging Requirements”.

1.2. Web Applications

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

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

Web applications are described in more detail in the Servlet 5.0 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 5.0 specification.

JSP 3.0 requires that these resources be implicitly associated with and accessible through a unique `ServletContext` instance available as the implicit `application` object (see Section 1.8, “Objects”).

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

- The `include` directive (see Section 1.10.3, “The `include` Directive”).
- The `taglib` directive (see Section 1.10.2, “The `taglib` Directive”).
- The `jsp:include` action element (see Section 5.4, “<jsp:include>”).
- The `jsp:forward` action (seeSection 5.5, “<jsp:forward>”).

JSP 3.0 supports portable packaging and deployment of web applications through the Servlet 5.0 specification. The Jakarta Server Pages specification inherits from the servlet specification the concepts of applications, `ServletContexts`, `Sessions`, `Requests` and `Responses`.

1.2.1. Relative URL Specifications

Elements may use relative URL specifications, called URI paths in the Servlet 5.0 specification. These paths are as described in RFC 3986. We refer to the path part of that specification, not the scheme, nor authority parts. Some examples are:

- A context-relative path is a path that starts with a slash (/). It is to be interpreted as relative to the

application to which the JSP page or tag file belongs. That is, its `ServletContext` object provides the base context URL.

- A page relative path is a path that does not start with a slash (/). It is to be interpreted as relative to the current JSP page, or the current JSP file or tag file, depending on where the path is being used. For an `include` directive (see Section 1.10.3, "The `include` Directive") where the path is used in a `file` attribute, the interpretation is relative to the JSP file or tag file. For a `jsp:include` action (see Section 5.4, "<jsp:include>") 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 1.10.5, "Including Data in JSP Pages" for exact details on this interpretation.

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

1.3. Syntactic Elements of a JSP Page

This section describes the basic syntax rules of JSP pages.

1.3.1. Elements and Template Data

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

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

1.3.2. Element Syntax

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

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

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

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

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

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

And:

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

An element has an element type describing its tag name, its valid attributes and its semantics. We refer to the type by its tag name.

JSP tags are case-sensitive, as in XML and XHTML.

An action may create objects and may make them available to the scripting elements through scripting-specific variables.

Scripting Elements
Scripting elements provide “glue” around template text and actions.

The Expression Language (EL) can be used to simplify accessing data from different sources. EL expressions can be used in JSP standard and custom actions and template data. EL expressions use the syntax `${expr}` and `#{expr}`. For example:

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

Chapter 2, *Expression Language* provides more details on the EL.

There are three language-based types of scripting elements: declarations, scriptlets, and expressions. Declarations follow the syntax `<%! ... %>`. Scriptlets follow the syntax `<% ... %>`. Expressions follow the syntax `<%= ... %>`.

1.3.3. Start and End Tags

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

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

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

1.3.4. Empty Elements

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

As examples, the following are all empty tags:

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

While the following are all non-empty tags:

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

1.3.5. Attribute Values

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

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

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

As of JSP 2.0, page authors can provide input in new ways using the `<jsp:attribute>` standard action element. Based on the configuration of the action being invoked, the body of the element either specifies a value that is evaluated once, or it specifies a "JSP fragment", which represents the body in a form that makes it possible for a tag handler to evaluate it as many times as needed. The `<jsp:attribute>` action must only be used to specify an attribute value for standard or custom actions.

A translation error must occur if it is used in any other context, for example to specify the value of template text that looks like an XML element.

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

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

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

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

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

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

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

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

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

1.3.7. Valid Names for Actions and Attributes

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

Attribute names that start with `jsp`, `_jsp`, `java` and `jakarta` are reserved in this specification.

1.3.8. White Space

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

This specification follows the whitespace behavior defined for XML. White space within the body text of a document is not significant, but is preserved. This default behavior can be modified for JSP pages in standard syntax as described in Section 3.3.8, “Removing Whitespaces from Template Text”.

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

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

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

The result is:

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

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

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

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

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

The result is:

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

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

It is possible to have extraneous whitespaces removed from template text through element `trim-directive-whitespaces` of JSP Property Groups (See Section 3.3.8, "Removing Whitespaces from Template Text"), or the page and tag file directive attribute `trimDirectiveWhitespaces` (See Section 1.10.1, "The `page` Directive", Section 8.5.1, "The tag Directive").

1.3.9. JSP Documents

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

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

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

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

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

1.3.10. JSP Syntax Grammar

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

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

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

In addition, the following notes and rules apply:

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

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

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

  - The `name` attribute appears exactly once and matches the `NonRTAttribute` production.
  - The `value` attribute appears at most once. If it appears, the `Attribute` production must be matched.
  - The `repeat` attribute appears exactly once and matches the `Attribute` production.

- There must be no other attributes aside from `name`, `value`, or `repeat`.

For example, the following sample strings match the above:

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

1.3.10.1. EBNF Grammar for JSP Syntax

```
JSPPage                   ::= Body

JSPTagDef                 ::= Body

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

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

ScriptlessBody            ::= (   ( '<%--'             JSPCommentBody      )
                                | ( '<%@'              DirectiveBody       )
                                | ( '<jsp:directive.'  XMLDirectiveBody    )
                                | ( '<%!'              <TRANSLATION_ERROR> )
                                | ( '<jsp:declaration' <TRANSLATION_ERROR> )
                                | ( '<%='              <TRANSLATION_ERROR> )
                                | ( '<jsp:expression'  <TRANSLATION_ERROR> )
                                | ( '<%'               <TRANSLATION_ERROR> )
                                | ( '<jsp:scriptlet'   <TRANSLATION_ERROR> )
                                | ( '${'               ELExpressionBody    )
                                | ( '#{'               ELExpressionBody    )
```

```
                              | ( '<jsp:text'        XMLTemplateText     )
                              | ( '<jsp:'           StandardAction      )
                              ( ( '</'              ExtraClosingTag     )
                              | ( '<'               CustomAction
                                                    CustomActionBody    )
                              | TemplateText
                            )*
                            [ vc: ELEnabled ]

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

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

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

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

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

XMLJSPDirectiveBody       ::=   S?
                              (   ( 'page' S PageDirectiveAttrList S?
                                     ( '/>' | ( '>' S? ETag ) )
                                  )
```

```
                                  | ( 'include' S IncludeDirectiveAttrList S?
                                      ( '/>' | ( '>' S? ETag ) )
                                    )
                                )
                              | <TRANSLATION_ERROR>

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

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

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

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

IncludeDirectiveAttrList   ::= ATTR[ !file ]

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

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

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

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

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

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

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

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

ELExpression               ::= [See EL spec document, production Expression]

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

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

StandardAction             ::=   ( 'useBean'     StdActionContent )
```

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

StdActionContent           ::= Attributes StdActionBody
                                [ vc: StdActionAttributesValid ]

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

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

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

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

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

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

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

OptionalBody               ::= EmptyBody | ActionBody

ScriptlessOptionalBody     ::= EmptyBody | ScriptlessActionBody

TagDependentOptionalBody   ::= EmptyBody | TagDependentActionBody
```

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

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

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

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

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

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

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

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

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

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

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

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

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

Param                       ::= '<jsp:param' StdActionContent

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

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

TagPrefix                   ::= Name

CustomActionName            ::= Name

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

CustomActionEnd             ::=   CustomActionTagDependent
                                | CustomActionJSPContent
                                | CustomActionScriptlessContent

CustomActionTagDependent    ::= TagDependentOptionalBody
                                [vc: CustomActionTagDependentMatch]
```

```
CustomActionJSPContent     ::= OptionalBody
                               [ vc: CustomActionJSPContentMatch ]

CustomActionScriptlessContent ::= ScriptlessOptionalBody
                                  [ vc: CustomActionScriptlessContentMatch ]

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

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

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

ExtraClosingTag            ::= ETag
                               [ vc: ExtraClosingTagMatch ]`

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

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

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

AnyAttributeValue          ::= AttributeValue | RTAttributeValue
```

```
AttributeValue              ::= AttributeValueDouble | AttributeValueSingle

RTAttributeValue            ::= RTAttributeValueDouble | RTAttributeValueSingle

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

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

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

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

Name                        ::= XML::Name

Char                        ::= XML::Char

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

S                           ::= XML::S

Eq                          ::= XML::Eq

CDSect                      ::= XML::CDSect
```

1.3.10.2. Validity Constraints

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

- `ScriptingEnabled` - The `ScriptlessBody` production must be followed if scripting is disabled for this translation unit. See the `scripting-invalid` JSP Configuration element (Section 3.3.3, "Disabling Scripting Elements").
- `ScriptlessBody` - The `AllBody` production cannot be followed if one of the parent nodes in the parse tree is a `ScriptlessBody` production. That is, once we have followed the `ScriptlessBody` production, until that production is complete we cannot choose the `AllBody` production.
- `ELEnabled` - The token `${` or `#{` is not followed if expressions are disabled for this translation unit. See the `isELIgnored` page and tag directive (See Section 1.10.1, "The `page` Directive") and Section 8.5.1, "The `tag` Directive" respectively) and the `el-ignored` JSP Configuration element (Section 3.3.2, "Deactivating EL Evaluation").
- `TagFileSpecificDirectives` - The `JSPDirectiveBody` production must be followed if the root production is `JSPPage` (i.e. this is a JSP page). The `TagDefDirectiveBody` production must be followed if the root production is `JSPTagDef` (i.e. this is a tag file).
- `TagFileSpecificXMLDirectives` - The `XMLJSPDirectiveBody` production must be followed if the root production is `JSPPage` (i.e. this is a JSP page). The `XMLTagDefDirectiveBody` production must be followed if the root production is `JSPTagDef` (i.e. this is a tag file).
- `PageDirectiveUniqueAttr` - A translation error will result if there is more than one occurrence of any attribute defined by this directive in a given translation unit, and if the value of the attribute is different than the previous occurrence. No translation error results if the value is identical to the previous occurrence. In addition, the `import` and `pageEncoding` attributes are excluded from this constraint (see Section 1.10.1, "The `page` Directive").
- `TagLibDirectiveUniquePrefix` - A translation error will result if the prefix `AttributeValue` has already previously been encountered as a potential `TagPrefix` in this translation unit.
- `TagDirectiveUniqueAttr` - A translation error will result if the prefix of this tag directive is already defined in the current scope, and if that prefix is bound to a namespace other than that specified by the `uri` or `tagdir` attribute.
- `UniqueAttributeName` - A translation error will result if there are two or more `attribute` directives with the same value for the `name` attribute in the same translation unit. A translation error will result if there is a `variable` directive with a `name-given` attribute equal to the value of the `name` attribute of an `attribute` directive in the same translation unit.
- `UniqueVariableName` - A translation error must occur if more than one `variable` directive appears in the same translation unit with the same value for the `name-given` attribute or the same value for the `name-from-attribute` attribute. A translation error must occur if there is a `variable` directive with a `name-given` attribute equal to the value of the `name` attribute of an `attribute` directive in the same translation unit. A translation error must occur if there is a `variable` directive with a `name-from-attribute` attribute whose value is not equal to the `name` attribute of an `attribute` directive in the same translation unit that is also of type `java.lang.String`, that is `required` and that is not an

rtexprvalue. A translation error must occur if the value of the alias attribute is equal to the value of a name-given attribute of a variable directive, or the value of the name attribute of an attribute directive in the same translation unit.

- TagFileSpecificActions - The invoke and doBody standard actions are only matched if the JSPTagDef production was followed (i.e. if this is a tag file instead of a JSP page).
- RTAttributeScriptingEnabled - If the RTAttributeValueDouble or RTAttributeValueSingle productions are visited during parsing and scripting is disabled for this page, a translation error must be produced. See the scripting-invalid JSP Configuration element (Section 3.3.3, “Disabling Scripting Elements”).
- ExpressionBodyContent - A translation error will result if the body content minus the closing delimiter (%> , or </jsp:expression>, depending on how the expression started) does not represent a well-formed expression in the scripting language selected for the JSP page.
- StdActionAttributesValid - An attribute is considered “provided” for this standard action if either the Attribute production or the AttributeBody production is followed in the context of the enclosing StandardAction production. A translation error will result if any of the following conditions is true:
  - The set of attributes “provided” for this standard action does not match one of the valid attribute combinations specified in **Table JSP.1-5** , “Valid body content and attributes for Standard Actions”.
  - The same attribute is “provided” more than once, as determined by the attribute name.
  - An attribute is “provided” using the AttributeBody production that does not accept a request-time expression value, as indicated by the = prefix in **Table JSP.1-5** , “Valid body content and attributes for Standard Actions”.
- StdActionBodyMatch - The StdActionBody production will only be matched if the production listed for this standard action in the “Body Production” column in **Table JSP.1-5** , “Valid body content and attributes for Standard Actions” is followed.
- AttributeBodyMatch - The type of element being specified determines which production is followed (see Section 5.10, “<jsp:attribute>” for details):
  - If a custom action that specifies an attribute of type JspFragment, ScriptlessBody must be followed.
  - If a standard or custom action that accepts a request-time expression value, AllJspBody must be followed.
  - If a standard or custom action that does not accept a request-time expression value, TemplateTextBody must be followed.
- JspBodyBodyContent - The ScriptlessBody production must be followed if the body content for this tag is scriptless. The Body production must be followed if the body content for this tag is JSP. The TagDependentBody production must be followed if the body content for this tag is tagdependent.
- UniqueAttSpec - A translation error will result if the same attribute name appears more than once.
- CustomActionMatchesAndValid - Following the rules in Section 7.3, “The Tag Library Descriptor” for determining the relevant set of tags and tag libraries, assume the following:

- Let `U` be the URI indicated by the `uri AttributeValue` of the previously encountered `TagLibDirectiveAttrList` with `prefix` matching the `TagPrefix` for this potential custom action, or `nil` if no such `TagLibDirectiveAttrList` was encountered in this translation unit.
- If `U` is not `nil`, let `L` be the `<taglib>` element in the relevant TLD entry such that `L.uri` is equal to `U`.

Then:

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

- `CustomActionTagDependentMatch` - Assume the definition of `L` from the `CustomActionMatchesAndValid` validity constraint above. The `CustomActionTagDependent` production is not matched if there does not exist a `<tag>` element `T` in a relevant TLD such that `L.T.body-content` contains the value `tagdependent`.
- `CustomActionJSPContentMatch` - Assume the definition of `L` from the `CustomActionMatchesAndValid` validity constraint above. The `CustomActionJSPContent` production is not matched if there exists a `<tag>` element `T` in a relevant TLD such that `L.T.body-content` does not contain the value `JSP`.
- `CustomActionScriptlessContentMatch` - Assume the definition of `L` from the `CustomActionMatchesAndValid` validity constraint above. The `CustomActionScriptlessContent` production is not matched if there does not exist a `<tag>` element `T` in a relevant TLD such that `L.T.body-content` contains the value `scriptless`.
- `QuotedDollarMatch` - The `\$` or `\#` token is only matched if EL is enabled for this translation unit. See Section 3.3.2, “Deactivating EL Evaluation”.
- `ETagMatch` - Assume the definition of `U` from the `CustomActionMatchesAndValid` validity constraint. If `TagPrefix` is not ‘`jsp`’ and `U` is `nil`, the `ETag` production is not matched. Otherwise, the `ETag` production is matched and a translation error will result if the prefix and name of this closing tag does not match the prefix and name of the starting tag at the corresponding nesting level, or if there is no corresponding nesting level (i.e. too many closing tags). This is similar to the way XML is defined, except that template text that looks like a closing element with an unrecognized prefix is allowed in the body of a custom or standard action. In the following example, assuming ‘my’ is a valid prefix and ‘indent’ is a valid tag, the `</ul>` tag is considered template text, and no translation error is produced:

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

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

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

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

1.3.10.3. Standard Action Attributes

Table JSP.1-5 , "Valid body content and attributes for Standard Actions" specifies, for each standard action element, the bodies and the attribute combinations that are valid. The value in the "Body Production" column specifies a production name that must be matched for the body of the standard action to be considered valid. The value in the "Valid Attribute Combinations" column uses the same syntax as the `attrset` notation described at the start of Section 1.3.10, "JSP Syntax Grammar", and indicates which attributes can be provided.

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

Element	Body Production	Valid Attribute Combinations

`jsp:useBean`	`OptionalBody`	`( !id, scope, !class )`
		`( !id, scope, !type )`
		`( !id, scope, !class, !type )`
		`( !id, scope, =!beanName, !type )`

`jsp:setProperty`	`EmptyBody`	`( !name, !property, param )` `( !name, !property, =!value )`
`jsp:getProperty`	`EmptyBody`	`( !name, !property )`
`jsp:include`	`ParamBody`	`( =!page, flush )`
`jsp:forward`	`ParamBody`	`( =!page )`

`jsp:plugin`	`PluginBody`	`( !type, !code, !codebase, align, archive,`
		`=height, hspace, jreversion, name, vspace,`
		`title, =width, nspluginurl, iepluginurl,`
		`mayscript )`

`jsp:invoke`	`EmptyBody`	`( !fragment, !var, scope )`
		`( !fragment, !varReader, scope )`
		`( !fragment )`

Element	Body Production	Valid Attribute Combinations
jsp:doBody	EmptyBody	(!var, scope) (!varReader, scope) ()
jsp:element	OptionalBody	(=!name)
jsp:output	EmptyBody	(omit-xml-declaration) (omit-xml-declaration, !doctype-root-element, !doctype-system, doctype-public)
jsp:param	EmptyBody	(!name, =!value)

1.4. Error Handling

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

1.4.1. Translation Time Processing Errors

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

1.4.2. Request Time Processing Errors

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

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.

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

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

page (or the implementation default behavior, if none is specified).

Information about the error is passed as `jakarta.servlet.ServletRequest` attributes to the error handler, with the same attributes as specified by the Servlet specification. Names starting with the prefixe `jakarta` are reserved by the different specifications of the Jakarta EE platform. The `jakarta.servlet` prefix is reserved and used by the servlet and JSP specifications.

1.4.3. Using JSPs as Error Pages

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

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

By default, a JSP error page sets the status code of the response to the value of `${pageContext.errorData.statusCode}` (which is equal to 500 by default), but may set it to a different value (including 200) as it sees fit.

A JSP container must detect if a JSP error page is self-referencing and throw a translation error.

1.5. Comments

There are different types of comments available in JSP pages in standard syntax and JSP documents (in XML syntax).

1.5.1. Comments in JSP Pages in Standard Syntax

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.

1.5.1.1. Generating Comments in Output to Client

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

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

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

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

1.5.1.2. 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 ... **/ %>
```

1.5.2. Comments in JSP Documents

Comments in JSP documents use the XML syntax, as follows:

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

The body of the content is ignored completely. Comments in JSP documents may be used for documentation purposes and for “commenting out” portions of a JSP page.

Comments in JSP documents do not nest.

1.6. Quoting and Escape Conventions

The following quoting conventions apply to JSP pages.

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

Quoting in EL Expressions

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

Quoting in Scripting Elements

- A literal `%>` is quoted by `%\>`

Quoting in Template Text

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

Quoting in Attributes

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

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

Examples

The following line shows an illegal attribute value.

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

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

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

The next lines are all legal quotations.

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

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

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

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

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

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

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

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

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

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

XML Documents

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

1.7. Overall Semantics of a JSP Page

A JSP page implementation class defines a `_jspService()` method mapping from the request to the response object. Some details of this transformation are specific to the scripting language used (see

Chapter 9, *Scripting*). 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 1.8.3, "Implicit Objects"). Its value varies:

- Initially, `out` is a new `JspWriter` object. This object may be different from the stream object returned from `response.getWriter()`, and may be considered to be interposed on the latter in order to implement buffering (see Section 1.10.1, "The `page` Directive"). This is the initial `out` object. JSP page authors are prohibited from writing directly to either the `PrintWriter` or `OutputStream` associated with the `ServletResponse`.
- The JSP container should not invoke `response.getWriter()` until the time when the first portion of the content is to be sent to the client. This enables a number of uses of JSP, including using JSP as a language to "glue" actions that deliver binary content, or reliably forwarding to a servlet, or change dynamically the content type of the response before generating content. See Chapter 4, *Internationalization Issues*.
- Within the body of some actions, `out` may be temporarily re-assigned to a different (nested) instance of a `JspWriter` object. Whether this is the case depends on the details of the action's semantics. Typically the content of these temporary streams is appended to the stream previously referred to by `out`, and `out` is subsequently re-assigned to refer to the previous (nesting) stream. Such nested streams are always buffered, and require explicit flushing to a nesting stream or their contents will be discarded.
- If the initial `out JspWriter` object is buffered, then depending upon the value of the `autoFlush` attribute of the `page` directive, the content of that buffer will either be automatically flushed out to the `ServletResponse` output stream to obviate overflow, or an exception shall be thrown to signal buffer overflow. If the initial `out JspWriter` is unbuffered, then content written to it will be passed directly through to the `ServletResponse` output stream.

A JSP page can also describe what should happen when some specific events occur. In JSP 3.0, the only events that can be described are the initialization and the destruction of the page. These events are described using "well-known method names" in declaration elements. (See Chapter 11, *JSP Container*).

1.8. Objects

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

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

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

1.8.1. Objects and Variables

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

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

The exact rules for the visibility of the variables are scripting language specific. Section 1.1, “What Is a JSP Page” defines the rules for when the `language` attribute of the `page` directive is `java`.

1.8.2. Objects and Scopes

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

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

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

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

- `page` - Objects with `page` scope are accessible only within the page where they are created. All references to such an object shall be released after the response is sent back to the client from the JSP page or the request is forwarded somewhere else. References to objects with `page` scope are stored in the `pageContext` object.
- `request` - Objects with `request` scope are accessible from pages processing the same request where they were created. References to the object shall be released after the request is processed. In particular, if the request is forwarded to a resource in the same runtime, the object is still reachable. References to objects with `request` scope are stored in the `request` object.
- `session` - Objects with `session` scope are accessible from pages processing requests that are in the same session as the one in which they were created. It is not legal to define an object with session scope from within a page that is not session-aware (see Section 1.10.1, “The `page` Directive”). All references to the object shall be released after the associated session ends. References to objects

with `session` scope are stored in the `session` object associated with the page activation.

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

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

1.8.3. Implicit Objects

JSP page authors have access to certain implicit objects that are always available for use within scriptlets and scriptlet expressions through scripting variables that are declared implicitly at the beginning of the page. All scripting languages are required to provide access to these objects. See Section 2.4, “Implicit Objects” for the implicit objects available within EL expressions. Implicit objects are available to tag handlers through the pageContext object, see below.

Each implicit object has a class or interface type defined in a core Java technology or Jakarta Servlet API package, as shown in **Table JSP.1-6** , “Implicit Objects Available in JSP Pages”.

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

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

Variable Name	Type	Semantics & Scope
`out`	`jakarta.servlet.jsp.JspWriter`	An object that writes into the output stream. `page` scope.
`config`	`jakarta.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. `page` scope. *When the scripting language is `java` then `page` is a synonym for `this` in the body of the page.*

In addition, the `exception` implicit object can be accessed in an error page, as described in **Table JSP.1-7 , "Implicit Objects Available in Error Pages"**.

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

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

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

See Section 7.5.1, "How to Define New Implicit Objects" for some non-normative conventions for the introduction of new implicit objects.

1.8.4. The pageContext Object

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

See the `jakarta.servlet.jsp` Javadocs for more details.

1.9. Template Text Semantics

The semantics of template (or uninterpreted) Text is very simple: the template text is passed through to the current `out JspWriter` implicit object, after applying the substitutions of Section 1.6, "Quoting and

Escape Conventions".

1.10. Directives

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

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

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

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

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

There are three directives: the `page`, the `taglib` and the `include` directive which are described next.

1.10.1. The `page` Directive

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

This `<jsp:directive.page>` element (Section 6.3.4, "The jsp:directive.page Element") describes the same information following the XML syntax.

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

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

Unrecognized attributes or values result in fatal translation errors.

Examples

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

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

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

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

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

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

Syntax

```
<%@ page page_directive_attr_list %>

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

The details of the attributes are as follows:

Table JSP.1-8 *Page Directive Attributes*

`language`	Defines the scripting language to be used in the scriptlets, expression scriptlets, and declarations within the body of the translation unit (the JSP page and any files included using the `include` directive below). In JSP 3.0, the only defined and required scripting language value for this attribute is `java` (all lowercase, case-sensitive). 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 9, *Scripting*. 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 1.8.3, "Implicit Objects". All scripting languages must support the Java Runtime Environment (JRE). All scripting languages must expose the Java technology object model to the script environment, especially implicit variables, JavaBeans component properties, and public methods. Future versions of the JSP specification may define additional values for the language attribute and all such values are reserved. It is a fatal translation error for a directive with a non- `java` language attribute to appear after the first scripting element has been encountered. Default is `java`.
`extends`	The value is a fully qualified Java programming language class name, that names the superclass of the class to which this JSP page is transformed (see Chapter 11, *JSP Container*). 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 7.5.1, "How to Define New Implicit Objects" for an alternate way to introduce objects into a JSP page that does not have this drawback.

`import`	An `import` attribute describes the types that are available to the scripting environment. The value is as in an import declaration in the Java programming language, 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. Packages `java.lang.*`, `jakarta.servlet.*`, `jakarta.servlet.jsp.*`, and `jakarta.servlet.http.*` are imported implicitly by the JSP container. No other packages may be part of this implicitly imported list. Page authors may use the include-prelude feature (see Section 3.3.5, "Defining Implicit Includes") in order to have additional packages imported transparently into their pages. This attribute is currently only defined when the value of the `language` directive is `java`.
`session`	Indicates that the page requires participation in an (HTTP) session. If `true` then the implicit script language variable named `session` of type `jakarta.servlet.http.HttpSession` references the current/new session for the page. If `false` then the page does not participate in a session; the `session` implicit variable is unavailable, and any reference to it within the body of the JSP page is illegal and shall result in a fatal translation error. Default is `true`.
`buffer`	Specifies the buffering model for the initial `out JspWriter` to handle content output from the page. If `none`, then there is no buffering and all output is written directly through to the `ServletResponse PrintWriter`. The size can only be specified in kilobytes. The suffix `kb` is mandatory or a translation error must occur. If a buffer size is specified then output is buffered with a buffer size not less than that specified. Depending upon the value of the `autoFlush` attribute, the contents of this buffer is either automatically flushed, or an exception is raised, when overflow would occur. The default is buffered with an implementation buffer size of not less than `8kb`. The corresponding JSP configuration element is buffer (see Section 3.3.10, "Setting Default Buffer Size").
`autoFlush`	Specifies whether the buffered output should be flushed automatically (`true` value) when the buffer is filled, or whether an exception should be raised (`false` value) to indicate buffer overflow. It is illegal, resulting in a translation error, to set `autoFlush` to `false` when `buffer=none`. The default value is `true`.

`isThreadSafe`	 The Servlet 2.4 specification deprecated `SingleThreadModel`, which is the most common mechanism for JSP containers to implement isThreadSafe. Page authors are advised against using isThreadSafe, as the generated Servlet may contain deprecated code. Indicates the level of thread safety implemented in the page. If `false` then the JSP container shall dispatch multiple outstanding client requests, one at a time, in the order they were received, to the page implementation for processing. If `true` then the JSP container may choose to dispatch multiple outstanding client requests to the page simultaneously. Page authors using `true` must ensure that they properly synchronize access to the shared state of the page. Default is `true`. Note that even if the `isThreadSafe` attribute is `false` the JSP page author must ensure that accesses to any shared objects are properly synchronized., The objects may be shared in either the `ServletContext` or the `HttpSession`.
`info`	Defines an arbitrary string that is incorporated into the translated page, that can subsequently be obtained from the page's implementation of `Servlet.getServletInfo` method.
`isErrorPage`	Indicates if the current JSP page is intended to be the URL target of another JSP page's `errorPage`. If `true`, then the implicit script language variable `exception` is defined and its value is a reference to the offending Throwable from the source JSP page in error. If `false` then the `exception` implicit variable is unavailable, and any reference to it within the body of the JSP page is illegal and shall result in a fatal translation error. Default is `false.`

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 1.2.1, "Relative URL Specifications". 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 jakarta.servlet.jsp.jspException (for backwards-compatibility) and also jakarta.servlet.error.exception (for compatibility with the servlet specification). See Section 1.4.3, "Using JSPs as Error Pages" for more details). Note: if autoFlush=true then if the contents of the initial JspWriter has been flushed to the ServletResponse output stream then any subsequent attempt to dispatch an uncaught exception from the offending page to an errorPage may fail. If the page defines an error page via the page directive, any error pages defined in web.xml will not be used.
contentType	Defines the MIME type and the character encoding for the response of the JSP page, and is also used in determining the character encoding of the JSP page. Values are either of the form "TYPE" or "TYPE;charset=CHARSET" with an optional white space after the ";". "TYPE" is a MIME type, see the IANA registry at http://www.iana.org/assignments/media-types/index.html for useful values. "CHARSET", if present, must be the IANA name for a character encoding. The default value for "TYPE" is "text/html" for JSP pages in standard syntax, or "text/xml" for JSP documents in XML syntax. If "CHARSET" is not specified, the response character encoding is determined as described in Section 4.2, "Response Character Encoding". The corresponding JSP configuration element is default-content-type (see Section 3.3.9, "Declaring Default Content Type"). See Chapter 4, *Internationalization Issues* for complete details on character encodings.

`pageEncoding`	Describes the character encoding for the JSP page. The value is of the form "`CHARSET`", which must be the IANA name for a character encoding. For JSP pages in standard syntax, the character encoding for the JSP page is the charset given by the `pageEncoding` attriute if it is present, otherwise the charset given by the `contentType` attribute if it is present, otherwise "`ISO-8859-1`". For JSP documents in XML syntax, the character encoding for the JSP page is determined as described in section 4.3.3 and appendix F.1 of the XML specification. The `pageEncoding` attribute is not needed for such documents. It is a translation-time error if a document names different encodings in its XML prolog / text declaration and in the `pageEncoding` attribute. The corresponding JSP configuration element is `page-encoding` (see Section 3.3.4, "Declaring Page Encodings"). See Chapter 4, *Internationalization Issues* for complete details on character encodings.
`isELIgnored`	Defines whether EL expressions are ignored or recognized for this page and translation unit. If `true`, EL expressions (of the form `${...}` and `#{...}`) are ignored by the container. If `false`, EL expressions (of the form `${...}` and `#{...}`) are recognized when they appear in template text or action attributes. The corresponding JSP configuration element is `el-ignored` (see Section 3.3.2, "Deactivating EL Evaluation"). The default value varies depending on the `web.xml` version - see Section 2.5, "Deactivating EL Evaluation".
`deferredSyntax-AllowedAsLiteral`	Indicates if the character sequence #{ is allowed or not when used as a String literal in this page and translation unit. If false (the default value), a translation error occurs when the character sequence is used as a String literal. The corresponding JSP configuration element is `deferred-syntax-allowed-as-literal` (see Section 3.3.7, "Deferred Syntax (character sequence #{)"). See Section 3.4, "Backwards Compatibility with JSP 2.0" for more information.
`trimDirective-Whitespaces`	Indicates how whitespaces in template text should be handled. If true, template text that contains only whitespaces is removed from the output. The default is not to trim whitespaces. This attribute is useful to remove the extraneous whitespaces from the end of a directive that is not followed by template text. The corresponding JSP configuration element is `trim-directive-whitespaces` (see Section 3.3.8, "Removing Whitespaces from Template Text"). The attribute is ignored by JSP documents (XML syntax).

1.10.2. The `taglib` Directive

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

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

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

result.

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

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

See Chapter 7, *Tag Extensions* for more specification details. And see Section 7.2.3, "Tag Library Directive" for an implementation note.

Section 6.3.1, "Namespaces, Standard Actions, and Tag Libraries" describes how the functionality of this directive can be exposed using XML syntax.

Examples

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

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

Syntax

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

where the attributes are:

Table JSP.1-9

`uri`	Either an absolute URI or a relative URI specification that uniquely identifies the tag library descriptor associated with this prefix. The URI is used to locate a description of the tag library as indicated in Chapter 7, *Tag Extensions*.
`tagdir`	Indicates this prefix is to be used to identify tag extensions installed in the `/WEB-INF/tags/` directory or a subdirectory. An implicit tag library descriptor is used (see Section 8.4, "Packaging Tag Files" for details). A translation error must occur if the value does not start with `/WEB-INF/tags`. A translation error must occur if the value does not point to a directory that exists. A translation error must occur if used in conjunction with the `uri` attribute.

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

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

1.10.3. The `include` Directive

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

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

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

The `<jsp:directive.include>` element (Section 6.3.5, “The jsp:directive.include Element”) describes the same information following the XML syntax.

Examples

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

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

Syntax

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

1.10.4. Implicit Includes

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

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

1.10.5. Including Data in JSP Pages

Including data is a significant part of the tasks in a JSP page. Accordingly, the JSP 3.0 specification has two include mechanisms suited to different tasks. A summary of their semantics is shown in **Table JSP.1-10** , "Summary of Include Mechanisms in JSP 3.0".

Table JSP.1-10 *Summary of Include Mechanisms in JSP 3.0*

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

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

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

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

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

- The JSP container must detect the syntax for each JSP file individually and parse each JSP file

according to the syntax in which it is written.

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

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

1.10.6. Additional Directives for Tag Files

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

1.11. EL Elements

EL expressions can appear in template data and in attribute values. EL expressions are defined in more detail in Chapter 2, *Expression Language*.

EL expressions can be disabled through the use of JSP configuration elements and page directives; see Section 1.10.1, “The `page` Directive” and Section 3.3.2, “Deactivating EL Evaluation”.

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

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

1.12. Scripting Elements

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

JSP 2.0 added EL expressions as an alternative to scripting elements. These are described in more detail in Chapter 2, *Expression Language*. Note that scripting elements can be disabled through the use of the `scripting-invalid` element in the web.xml deployment descriptor (see Section 3.3.3, “Disabling

Scripting Elements”).

There are three other classes of scripting elements: declarations, scriptlets and expressions. The scripting language used in the current page is given by the value of the `language` directive (see Section 1.10.1, “The `page` Directive”). In JSP 3.0, the only value defined is `java`.

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

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

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

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

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 6.3.7, “Scripting Elements”.

1.12.1. Declarations

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

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

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

The `<jsp:declaration>` element (Section 6.3.7, "Scripting Elements") describes the same information following the XML syntax.

Examples

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

```
<%! int i; %>
```

```
<%! int i = 0; %>
```

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

Syntax

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

1.12.2. Scriptlets

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

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

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

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

The `<jsp:scriptlet>` element (Section 6.3.7, "Scripting Elements") describes the same information

following the XML syntax.

Examples

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

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

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

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

Syntax

```
<% scriptlet %>
```

1.12.3. Expressions

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

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

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

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

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

The `<jsp:expression>` element (Section 6.3.7, “Scripting Elements”) describes the same information following the XML syntax.

Examples

This example inserts the current date.

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

Syntax

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

1.13. Actions

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

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

New actions are defined according to the mechanisms described in Chapter 7, *Tag Extensions* and the `jakarta.servlet.jsp.tagext` Javadoc and are introduced using the `taglib` directive.

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

1.14. Tag Attribute Interpretation Semantics

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

Many values are fixed translation-time values, but JSP 3.0 also provides a mechanism for describing values that are computed at request time, the rules are described in Section 1.14.1, “Request Time Attribute Values”.

1.14.1. Request Time Attribute Values

An attribute value of the form `"<%= scriptlet_expr %>"` or `'<%= scriptlet_expr %>'` denotes a request-time attribute value. The value denoted is that of the scriptlet expression involved. If Expression Language evaluation is not deactivated for the translation unit (see Section 3.3.2, “Deactivating EL Evaluation”) then request-time attribute values can also be specified using the EL through the syntax `'${el_expr}'` or `"${el_expr}"` (as well as `'#{el_expr}'` or `"#{el_expr}"`). Containers must also recognize

multiple EL expressions mixed with optional string constants. For example, `"Version ${major}.${minor} Installed"` is a valid request-time attribute value.

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

Quotation is done as in any other attribute value (Section 1.6, “Quoting and Escape Conventions”).

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

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

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

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

1.14.2. Type Conversions

We describe two cases for type conversions.

1.14.2.1. Conversions from String values

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

String values can be used to assign values to a type that has a `PropertyEditor` class as indicated in the JavaBeans specification. When that is the case, the `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.1-11** , “Conversions from string values to target type”. The conversion applied is that shown in the table.

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

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

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

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

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

1.14.2.2. Conversions from request-time expressions

Request-time expressions can be assigned to properties of any type. In the case of scriptlet expressions, no automatic conversions will be performed. In the case of EL expressions, the rules in section 1.23, "Type Conversion" of the EL 4.0 specification document must be followed.

Chapter 2. Expression Language

Please consult the Jakarta Expression Language 4.0 specification document for details on the Expression Language supported by JSP 3.0.

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

The EL is available in attribute values for standard and custom actions and within template text.

This chapter describes how the expression language is integrated within the JSP 3.0 environment.

2.1. Syntax of Expressions in JSP Pages: ${} vs #{}

There are two constructs to represent EL expressions: `${expr}` and `#{expr}`. While the EL parses and evaluates `${}` and `#{}` the same way, additional restrictions are placed on the usage of these delimiters in JSP pages.

An EL expression that is evaluated immediately is represented in JSP with the syntax `${}`, while an EL expression whose evaluation is deferred is represented with the syntax `#{}`.

2.2. Expressions and Template Text

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

Only the `${}` syntax is allowed for expressions in template text. A translation error will result if `#{}` is used in template text unless `#{}` is turned off via a backwards compatibility mechanism.

All EL expressions in JSP template text are evaluated as Strings, and are evaluated by the JSP engine immediately when the page response is rendered.

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

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

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

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

2.3. Expressions and Attribute Values

EL expressions can be used in any attribute that can accept a run-time expression, be it a standard action or a custom action. For more details, see the sections on backward compatibility issues, specifically Section 2.5, “Deactivating EL Evaluation” and Section 2.6, “Disabling Scripting Elements”.

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

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

Note that the normal JSP coercion mechanism already allows for:

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

An EL expression that appears in an attribute value is processed differently depending on the attribute’s type category defined in the TLD. Details are provided in the sections below.

2.3.1. Static Attribute

- Defined in the TLD through element `<rtexprvalue>` set to `false`.
- Type is always `java.lang.String`.
- Value must be a String literal (since it is determined at translation time). It is illegal to specify an expression.
- Type in the TLD is ignored. The String value is converted to the attribute’s target type (as defined in the tag handler) using the conversions defined in **Table JSP.1-11** , “Conversions from string values to target type”.

2.3.2. Dynamic Attribute

- Defined in the TLD through element `<rtexprvalue>` set to `true`.
- If type is not specified in the TLD, defaults to `java.lang.Object`.
- Value can be a String literal, a scriptlet expression, or an EL expression using the `${}` syntax.
- An EL expression is parsed using `ExpressionFactory.createValueExpression()` (with an expected

type equal to the type specified in the TLD) and the evaluation of the expression takes place immediately by calling method `getValue()` on the `ValueExpression`. After evaluation of the expression, the value is coerced to the expected type. The resulting value is passed in to the setter method for the tag attribute.

2.3.3. Deferred Value

- Defined in the TLD through element `<deferred-value>`.
- If type is not specified in the TLD, defaults to `java.lang.Object`.
- Value can be a String literal or an EL expression using the `#{}` syntax.
- An EL expression is parsed using `ExpressionFactory.createValueExpression()` (with an expected type equal to the type specified in the TLD). The expression is not evaluated. The result of parsing the expression is passed directly to the setter method of the tag attribute, whose argument type must be `javax.el.ValueExpression`. This allows for deferred evaluation of EL expressions. When the expression is evaluated by the tag handler, the value is coerced to the expected type. If a static value is provided, it is converted to a `ValueExpression` where `isLiteralText()` returns `true`.

2.3.4. Deferred Method

- Defined in the TLD through element `<deferred-method>`.
- If the method signature is not defined in the TLD, it defaults to `void method()`.
- Value can be a String literal or an EL expression using the `#{}` syntax.
- An EL expression is parsed using `ExpressionFactory.createMethodExpression()` (with a method signature equal to the method signature specified in the TLD). The result of parsing the expression is passed directly to the setter method of the tag attribute, whose argument type must be `jakarta.el.MethodExpression`. This allows for deferred processing of EL expressions that identify a method to be invoked on an Object.
- A String literal can be provided, as long as the return type of the deferred method signature is not void. A `MethodExpression` is created, which when invoked, returns the String literal coerced to expected return type (the standard EL coercion rules - see section 1.23, "Type Conversion" of the EL 4.0 specification) apply. A translation error occurs if the return type is void or if the string literal cannot be coerced to the return type of the deferred method signature.

2.3.5. Dynamic Attribute or Deferred Expression

- Defined in the TLD through elements `<rtexprvalue>` (see Section 2.3.2, “Dynamic Attribute”) specified together with `<deferred-value>` (see Section 2.3.3, “Deferred Value”) or `<deferred-method>` (see Section 2.3.4, “Deferred Method”).
- Value can be a String literal, a scriptlet expression, or an EL expression using the `${}` or `#{}` syntax. The attribute value is considered a deferred value or a deferred method if the value is an EL expression using the `#{}` syntax. It is considered a dynamic attribute otherwise.

- The attribute value is processed according to its type category as described above. The only difference is that the setter method argument must be of type `java.lang.Object`. The setter method will normally use `instanceof` to discriminate whether the attribute value is a dynamic attribute or a deferred value.

2.3.6. Examples of Using ${} and #{}

As an example, assume a tag with the following three attributes:

- static - `rtexprvalue=false, type=java.lang.String`
- dynamic - `rtexprvalue=true, type=java.lang.String`
- deferred - `rtexprvalue=true, type=java.lang.ValueExpression`

The following tags would yield the following results:

Table JSP.2-1 *Examples of Using ${} and #{}*

Expression	Result
`<my:tag static="xyz" />`	OK
`<my:tag static="${x[y]}" />`	ERROR
`<my:tag static="#{x[y]}" />`	ERROR
`<my:tag dynamic="xyz" />`	OK
`<my:tag dynamic="${x[y]}" />`	OK
`<my:tag dynamic="#{x[y]}" />`	ERROR
`<my:tag deferred="xyz" />`	OK
`<my:tag deferred="${x[y]}" />`	ERROR
`<my:tag deferred="#{x[y]}" />`	OK

2.4. Implicit Objects

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

- `pageContext` - the `PageContext` object
- `pageScope` - a `Map` that maps page-scoped attribute names to their values
- `requestScope` - a `Map` that maps request-scoped attribute names to their values
- `sessionScope` - a `Map` that maps session-scoped attribute names to their values
- `applicationScope` - a `Map` that maps application-scoped attribute names to their values
- `param` - a `Map` that maps parameter names to a single String parameter value (obtained by calling `ServletRequest.getParameter(String name)`)

- `paramValues` - a `Map` that maps parameter names to a `String[]` of all values for that parameter (obtained by calling `ServletRequest.getParameterValues(String name)`)
- `header` - a `Map` that maps header names to a single String header value (obtained by calling `HttpServletRequest.getHeader(String name)`)
- `headerValues` - a `Map` that maps header names to a `String[]` of all values for that header (obtained by calling `HttpervletRequest.getHeaders(String)`)
- `cookie` - a `Map` that maps cookie names to a single `Cookie` object. Cookies are retrieved according to the semantics of `HttpServletRequest.getCookies()`. If the same name is shared by multiple cookies, an implementation must use the first one encountered in the array of `Cookie` objects returned by the `getCookies()` method. However, users of the cookie implicit object must be aware that the ordering of cookies is currently unspecified in the servlet specification.
- `initParam` - a `Map` that maps context initialization parameter names to their `String` parameter value (obtained by calling `ServletContext.getInitParameter(String name)`)

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

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

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

2.5. Deactivating EL Evaluation

Since the syntactic patterns `${ expr }` and `#{ expr }` were not reserved in the JSP specifications before JSP 2.0, there may be situations where such patterns appear but the intention is not to activate EL expression evaluation but rather to pass through the pattern verbatim. To address this, the EL evaluation machinery can be deactivated as indicated in Section 3.3.2, "Deactivating EL Evaluation".

2.6. Disabling Scripting Elements

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

2.7. Invalid EL Expressions

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

2.8. Errors, Warnings, Default Values

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

2.9. Resolution of Variables and their Properties

The EL API provides a generalized mechanism, an `ELResolver`, implemented by the JSP container and which defines the rules that govern the resolution of variables and object properties.

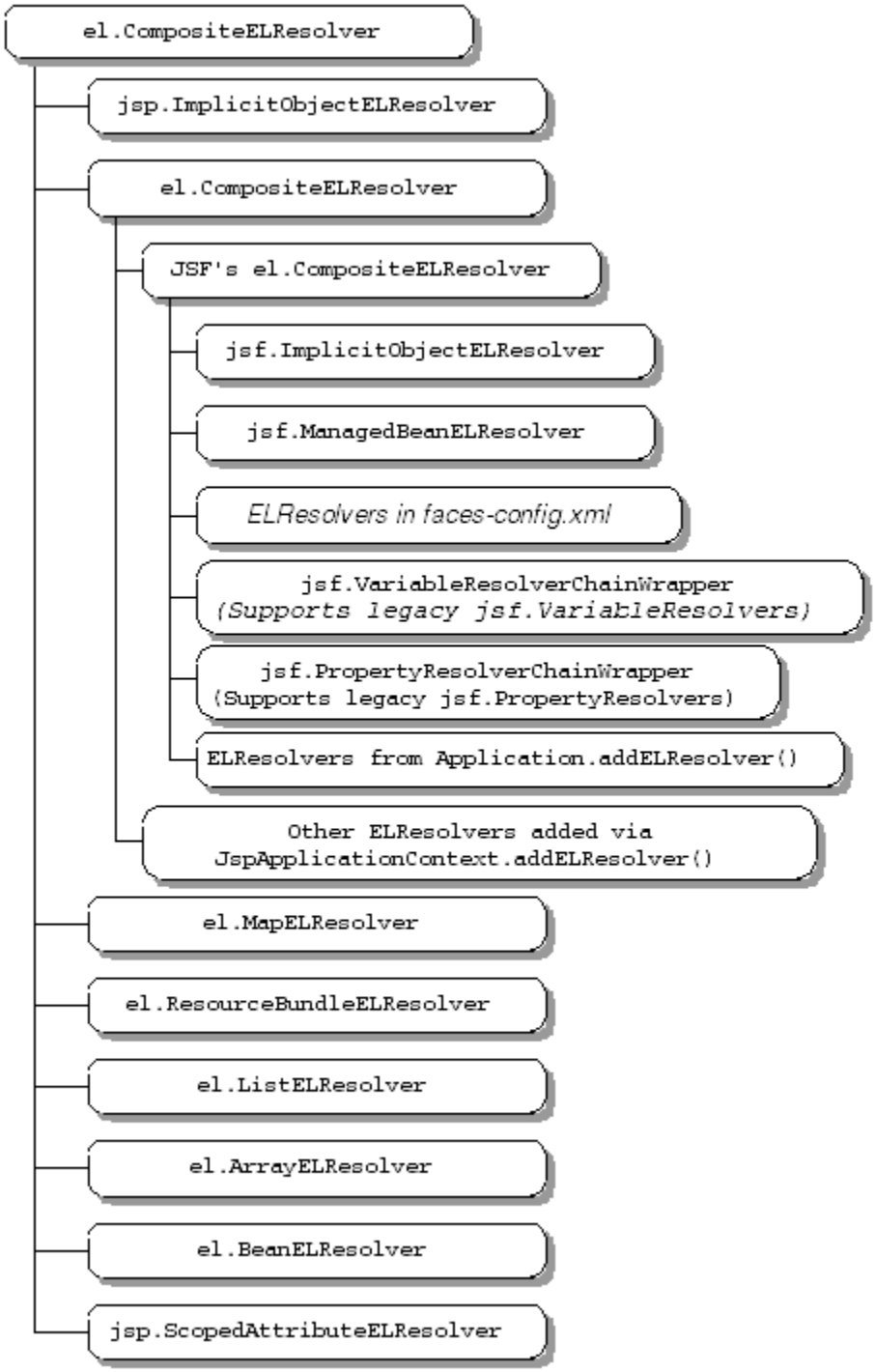
The `ELResolver` shown in **Figure JSP.2-1** JSP Resolver Hierarchy is passed to all EL expressions that appear in a JSP page or tag file. It is an instance of `jakarta.el.CompositeELResolver` that contains the following component `ELResolver`s, in order:

1. `jsp.ImplicitObjectELResolver`
   Resolves the implicit objects mentioned in Section 2.4, “Implicit Objects”.
2. All `ELResolver`s added via `JspApplicationContext.addELResolver()`, in the same order in which they were registered.
   This itself can take the form of a `el.CompositeELResolver`. This will include the `ELResolver` registered by Faces.
3. The `ELResolver` returned by `ExpressionFactory.getStreamELResolver()`.
4. `jakarta.el.StaticFieldResolver`
5. `jakarta.el.MapELResolver` - constructed in read/write mode.
6. `jakarta.el.ResourceBundleELResolver`
7. `jakarta.el.ListELResolver` - constructed in read/write mode.
8. `jakarta.el.ArrayELResolver` - constructed in read/write mode.
9. `jakarta.el.BeanELResolver` - constructed in read/write mode.
   Handles all cases where `base != null`
10. `jsp.ScopedAttributeELResolver`
    Handles all cases where `base == null`.
    Provides a map for other identifiers by looking up its value as an attribute, according to the behavior of `PageContext.findAttribute(String)` on the `pageContext` object. For example:

```
${product}
```

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

Figure JSP.2-1 *JSP Resolver Hierarchy*



2.10. Functions

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

Function mapping information is bound into the `ValueExpression` or `MethodExpression` at parse-time and is serialized along with the state of the expression. No function mapper needs to be provided at evaluation time.

2.10.1. Invocation Syntax

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

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

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

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

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

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

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

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

2.10.2. Tag Library Descriptor Information

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

Each such function is given a name (as seen in the EL), and a static method in a specific class that will

implement the function. The class specified in the TLD must be a public class, and must be specified using a fully-qualified class name (including packages). The specified method must be a public static method in the specified class, and must be specified using a fully-qualified return type followed by the method name, followed by the fully-qualified argument types in parenthesis, separated by commas (see the XML Schema for Tag Library Descriptors for a full description of this syntax). Failure to satisfy these requirements shall result in a translation-time error.

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

Reference the `function` element in the XML Schema for Tag Library Descriptors for how to specify a function in the TLD.

2.10.3. Example

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

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

The following EL fragment shows the invocation of the function:

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

2.10.4. Semantics

- If the function has no prefix, the default namespace is used and the function is not validated. If the function has a prefix, assume the namespace as that associated with the prefix.

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

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

public static method with name and signature equal to the value of the `function-signature` element. If any of these don't exist, a translation-time error shall occur.

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

The introduction in Expression Language 3.0 of Lambdas and the ability to import methods at runtime via an `ImportHandler`, mean it is no longer possible to validate functions without a prefix at translation time.

Chapter 3. JSP Configuration

This chapter describes the JSP configuration information, which is specified in the Web Application Deployment Descriptor in `WEB-INF/web.xml`. For Servlet 5.0, the Web Application Deployment Descriptor is defined using XML Schema, and imports the elements described in the XML Schema for JSP 3.0 Deployment Descriptor. See the XML Schema for the details on how to specify JSP configuration information in a Web Application.

3.1. JSP Configuration Information in web.xml

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

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

3.2. Taglib Map

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

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

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

A `taglib-location` element contains a resource location (as indicated in Section 1.2.1, “Relative URL Specifications”) of the Tag Library Description File for the tag library.

3.3. JSP Property Groups

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

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

The applicability of a JSP property group is defined through one or more URL patterns. URL patterns use the same syntax as defined in Chapter SRV.12 of the Servlet 5.0 specification, but are bound at translation time. All the properties in the group apply to the resources in the Web Application that match any of the URL patterns. There is an implicit property: that of being a JSP file. JSP Property Groups do not affect tag files.

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

3.3.1. JSP Property Groups

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

- Indicate that a resource is a JSP file (implicit).
- Control disabling of EL evaluation.
- Control disabling of Scripting elements.
- Indicate page Encoding information.
- Prelude and Coda automatic includes.
- Indicate that a resource is a JSP document.
- Indicate that the deferred syntax (initiated by the character sequence `#{`) is allowed as a String literal.
- Control handling of whitespaces in template text.
- Indicate response ContentType information.
- Indicate response buffer size.
- Control handling of undeclared namespaces in a JSP page.

3.3.2. Deactivating EL Evaluation

Since the syntactic pattern `${expr}` was not reserved in the JSP specifications before JSP 2.0, and the syntactic pattern `#{expr}` was not reserved before JSP 2.1, there may be situations where such patterns appear but the intention is not to activate EL expression evaluation but rather to pass through the pattern verbatim. To address this, the EL evaluation machinery can be deactivated as indicated in this section.

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

default to evaluating expressions.

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

The default mode for JSP pages in a Web Application delivered using a `web.xml` using the Servlet 2.4 or later format is to evaluate EL expressions with the ${} syntax. Expressions using the #{} are evaluated starting with JSP 2.1. See Section 3.4, "Backwards Compatibility with JSP 2.0" for more details on the evaluation of #{} expressions.

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

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

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

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

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

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

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

Table JSP.3-2 , "EL Evaluation Settings for Tag Files" summarizes the EL evaluation settings for tag files, and their meanings:

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

Tag Directive isELIgnored	EL Encountered
unspecified	Evaluated
false	Evaluated
true	Ignored

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

3.3.3. Disabling Scripting Elements

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

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

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

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

Table JSP.3-3 , "Scripting Settings" summarizes the scripting settings and their meanings:

Table JSP.3-3 *Scripting Settings*

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

3.3.4. Declaring Page Encodings

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

The `page-encoding` element is a subelement of `jsp-property-group` (see Section 3.3.1, “JSP Property Groups”). It has no subelements. Its valid values are those of the `pageEncoding` page directive. It is a translation-time error to name different encodings in the `pageEncoding` attribute of the page directive of a JSP page and in a JSP configuration element matching the page. It is also a translation-time error to name different encodings in the prolog / text declaration of the document in XML syntax and in a JSP configuration element matching the document. It is legal to name the same encoding through multiple mechanisms.

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

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

3.3.5. Defining Implicit Includes

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

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

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

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

```
<jsp-property-group>
  <url-pattern>*.jsp</url-pattern>
  <include-prelude>/WEB-INF/jspf/prelude1.jspf</include-prelude>
  <include-coda>/WEB-INF/jspf/coda1.jspf</include-coda>
</jsp-property-group>
<jsp-property-group>
  <url-pattern>/two/*</url-pattern>
  <include-prelude>/WEB-INF/jspf/prelude2.jspf</include-prelude>
  <include-coda>/WEB-INF/jspf/coda2.jspf</include-coda>
</jsp-property-group>
```

3.3.6. Denoting XML Documents

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

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

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

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

3.3.7. Deferred Syntax (character sequence #{)

As of JSP 2.1, the character sequence `#{` is reserved for EL expressions. Consequently, a translation error occurs if the `#{` character sequence is used as a String literal (in template text of a JSP 2.1+ container or as an attribute value for a tag-library where jsp-version is 2.1+).

The `deferred-syntax-allowed-as-literal` element is a subelement of `jsp-property-group` (See Section 3.3.1, "JSP Property Groups"). It has no subelements. Its valid values are `true` and `false`, and it is disabled (`false`) by default. Allowing the character sequence `#{` when used as a String literal can be done by setting the `deferred-syntax-allowed-as-literal` element to `true` in the JSP configuration.

Page authors can override the default value through the `deferredSyntaxAllowedAsLiteral` attribute of the page directive (see Section 1.10, "Directives"). See also Section 3.4, "Backwards Compatibility with JSP 2.0" for more information.

3.3.8. Removing Whitespaces from Template Text

Whitespaces in template text of a JSP page are preserved by default (See Section 1.3.8, "White Space"). Unfortunately, this means that unwanted extraneous whitespaces often make it into the response output.

For example, the following code snippet (where `<EOL>` represents the end-of-line character(s))

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %><EOL>
<%@ taglib prefix="x" uri="http://java.sun.com/jsp/jstl/xml" %><EOL>
Hello World!<EOL>
```

would generate the following output:

```
<EOL>
<EOL>
Hello World!<EOL>
```

For JSP pages (standard syntax), the JSP configuration element `trim-directive-whitespaces` can be used to indicate that template text containing only whitespaces must be removed from the response output. It has no effect on JSP documents (XML syntax). In the example above, the first `<EOL>` represents template text that contains only whitespaces and would therefore be removed. `<EOL>` HelloWorld! `<EOL>` represents template text that does not contain only whitespaces and would therefore be preserved as-is.

```
<EOL>
Hello World!<EOL>
```

The `trim-directive-whitespaces` element is a subelement of `jsp-property-group` (See Section 3.3.1, "JSP Property Groups"). It has no subelements. Its valid values are `true` and `false`, and it is disabled (`false`) by default. Enabling the trimming of whitespaces can be done by setting the `trim-directive-whitespaces` element to `true` in the JSP configuration.

Page authors can override the default value through the `trimDirectiveWhitespaces` attribute of the page directive (see Section 1.10, "Directives").

3.3.9. Declaring Default Content Type

The JSP configuration element `default-content-type` can be used to specify the default `contentType` property of a group of JSP pages defined using the `jsp-property-group` element.

The valid values for the `default-content-type` element are those of the `contentType` attribute of the `page` directive. It specifies the default response `contentType` if the `page` directive does not include a `contentType` attribute.

3.3.10. Setting Default Buffer Size

The JSP configuration element buffer can be used to specify the default buffering model for the initial out JspWriter for a group of JSP pages defined using the the jsp-property-group element.

The valid values for the buffer element are those of the buffer attribute of the page directive. It can be used to specify if buffering should be used for the output to Response, and if so, the size of the buffer to use.

3.3.11. Raising Errors for Undeclared Namespaces

The default behavior when a tag with unknown namespace is used in a JSP page is to silently ignore it. For most page authors, this is often a source of errors. To make the mistakes obvious, this JSP configuration element can be used to force an error when an unknown namespace is used in a JSP page, as is already the case for JSP documents (XML syntax).

The `error-on-undeclared-namespace` element is a subelement of `jsp-property-group`. It has no subelements. Its valid values are `true` and `false`, with false being the default.

If it is set to `true`, then an error must be raised during the translation time, when an undeclared tag is used in a JSP page.

3.4. Backwards Compatibility with JSP 2.0

As of JSP 2.1, the character sequence `#{` was reserved for EL expressions.

When used as a tag attribute value, the `#{expr}` syntax is evaluated by the container only if the the jsp-version element specified in the TLD has the value 2.1 or higher. If the version specified is less than 2.1, then the `#{expr}` syntax is simply processed as a String literal.

When used in template text in a JSP page, the `#{` character sequence triggers a translation error, unless specifically allowed through a configuration setup. This is because the `#{}` syntax is associated exclusively with deferred-evaluation in JSP 2.1 and does not make sense in the context of template text (only immediate evaluation using the `${expr}` syntax makes sense in template text).

In a tag file, `#{expr}` in template text is handled according to the tag file's JSP version. If the tag file's JSP version is 2.0 or less, `#{expr}` in template text will not cause any error. If the tag file's JSP version is equal to or greater than 2.1, `#{expr}` in template text must cause an error, unless it has been escaped or the tag file contains a `deferredSyntaxAllowedAsLiteral` tag directive attribute set to `TRUE`. See Section 8.4.2, "Packaging in a JAR", and Section 8.4.3, "Packaging Directly in a Web Application", for how the JSP version of a tag file is determined.

Chapter 4. Internationalization Issues

This chapter describes requirements for internationalization with Jakarta Server Pages.

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

Primarily, this specification addresses the issues of character encodings.

The Java programming language represents characters internally using the Unicode character encoding, which provides support for most languages. As of Java 8, the Unicode 6.2 character set is supported. For storage and transmission over networks, however, many other character encodings are used. The Java SE platform therefore also supports character conversion to and from other character encodings. Any Java runtime must support the Unicode transformations UTF-8, UTF-16BE, and UTF-16LE as well as the ISO-8859-1 (Latin-1) and US-ASCII character encodings, but most implementations support many more.

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

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

4.1. Page Character Encoding

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

4.1.1. Standard Syntax

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

- A byte order mark (BOM)
- A JSP configuration element `page-encoding` value whose URL pattern matches the page.
- The `pageEncoding` attribute of the `page` directive of the page. It is a translation-time error to name different encodings in the `pageEncoding` attribute of the page directive of a JSP page and in a JSP configuration element whose URL pattern matches the page.
- The charset value of the `contentType` attribute of the page directive. This is used to determine the page character encoding if neither a JSP configuration element `page-encoding` nor the `pageEncoding`

attribute are provided.

- If none of the above is provided, ISO-8859-1 is used as the default character encoding.

For tag files in standard syntax, the page character encoding is determined from a BOM or the `pageEncoding` attribute of the `tag` directive of the tag file (in this precedence order), or is `ISO-8859-1` if neither is specified.

A BOM consists of the Unicode character code `U+FEFF` at the beginning of a data stream, where it is used to define the byte order and encoding form of unmarked plaintext files.

The exact byte representation of the BOM depends on the particular encoding of the text file, as follows:

Table JSP.4-1 *Byte representations of the BOM*

Bytes	Encoding Form
`FE FF`	UTF-16, big-endian
`FF FE`	UTF-16, little-endian
`00 00 FE FF`	UTF-32, big-endian
`FF FE 00 00`	UTF-32, little-endian
`EF BB BF`	UTF-8

The above byte sequences have been reserved to identify a BOM at the beginning of JSP pages in standard syntax, and will not appear in the page's output.

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

When using a BOM, it is legal to describe the character encoding in a JSP configuration element page-encoding or a pageEncoding attribute of the page directive of the page, as long as they are consistent.

4.1.2. XML Syntax

For JSP documents and tag files in XML syntax, the page character encoding is determined as described in section 4.3.3 and appendix F.1 of the XML specification.

For JSP documents in XML syntax, it is legal to also describe the character encoding in a JSP configuration element `page-encoding` or a `pageEncoding` attribute of the page directive of the document, as long as they are consistent. It is a translation-time error to name different encodings in two or more of the following: the XML prolog / text declaration of a JSP document, the `pageEncoding` attribute of the

page directive of the JSP document, and in a JSP configuration element whose URL pattern matches the document.

Note that for tag files in XML syntax, it is illegal for the tag directive to include a pageEncoding attribute: the encoding is inferred solely by using the conventions for XML documents.

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

4.2. Response Character Encoding

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

The JSP container determines an initial response character encoding along with the initial content type for a JSP page and calls `ServletResponse.setContentType()` with this information before processing the page. JSP pages can set initial content type and initial response character encoding using the `contentType` attribute of the page directive. The JSP configuration element default-content-type can also be used to set the default initial content type and default initial response chrarcter encoding of a group of JSP pages using the jsp-property-group element. See Section 3.3.9, "Declaring Default Content Type".

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

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

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

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

Note that the response character encoding can only be changed until the response is committed. Data is sent to the response stream on buffer flushes for buffered pages, or on encountering the first content

(beware of whitespace) on unbuffered pages. Whitespace is notoriously tricky for JSP Pages in JSP syntax, but much more manageable for JSP documents in XML syntax.

4.3. Request Character Encoding

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

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

4.4. XML View Character Encoding

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

The XML view character encoding is always `UTF-8`.

4.5. Delivering Localized Content

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

Chapter 5. Standard Actions

This chapter describes the standard actions of Jakarta Server Pages 3.0 (JSP 3.0). Standard actions are represented using XML elements with a prefix of `jsp` (though that prefix can be redefined in the XML syntax). A translation error will result if the JSP prefix is used for an element that is not a standard action.

5.1. <jsp:useBean>

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

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

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

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

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

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

The id Attribute

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

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

  Duplicate `id` 's found in the same translation unit shall result in a fatal translation error.

- The JSP container will associate an object (a JavaBean component) with the named value and

accessed via that name in various contexts through the `pagecontext` object described later in this specification.

The `name` is also used to expose a variable (`name`) in the page's scripting language environment. The scope of the scripting language variable is dependent upon the scoping rules and capabilities of the scripting language used in the page.

Note that this implies the `name` value syntax must comply with the variable naming syntax rules of the scripting language used in the page. Chapter 9, *Scripting* provides details for the case where the language attribute is `java`.

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

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

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

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

The scope Attribute

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

Semantics

The actions performed in a `jsp:useBean` action are:

1. An attempt to locate an object based on the attribute values `id and scope`. For application and session scope, the inspection is done synchronized per scope namespace to avoid non-deterministic behavior.
2. A scripting language variable of the specified type (if given) or `class` (if `type` is not given) is defined with the given `id` in the current lexical scope of the scripting language. The `type` attribute should be used to specify a Java type that cannot be instantiated as a JavaBean (i.e. a Java type that is an abstract class, interface, or a class with no public no-args constructor). If the `class` attribute is used for a Java type that cannot be instantiated as a JavaBean, the container may consider the page invalid, and is recommended to (but not required to) produce a fatal translation error at translation time, or a `java.lang.InstantiationException` at request time. Similarly, if either `type` or `class` specify a type that can not be found, the container may consider the page invalid, and is recommended to (but not required to) produce a fatal translation error at translation time, or a `java.lang.ClassNotFoundException` at request time.
3. If the object is found, the variable's value is initialized with a reference to the located object, cast to the specified `type`. If the cast fails, a `java.lang.ClassCastException` shall occur. This completes the processing of this `jsp:useBean` action.
4. If the `jsp:useBean` action had a non-empty body it is ignored. This completes the processing of this `jsp:useBean` action.
5. If the object is not found in the specified scope and neither class nor beanName are given, a `java.lang.InstantiationException` shall occur. This completes the processing of this `jsp:useBean` action.
6. If the object is not found in the specified `scope`, and the `class` specified names a non-abstract class that defines a public no-args constructor, then the class is instantiated. The new object reference is associated with the scripting variable and with the specified name in the specified scope using the appropriate scope dependent association mechanism (see `PageContext`). After this, step 8 is performed.
   If the object is not found, and the `class` is either abstract, an `interface`, or no public no-args constructor is defined therein, then a `java.lang.InstantiationException` shall occur. This completes the processing of this `jsp:useBean` action.
7. If the object is not found in the specified `scope` ; and `beanName` is given, then the method `instantiate` of `java.beans.Beans` will be invoked with the `ClassLoader` of the servlet object and the `beanName` as arguments. If the method succeeds, the new object reference is associated the with the scripting variable and with the specified name in the specified scope using the appropriate scope dependent association mechanism (see `PageContext`). After this, step 8 is performed.
8. If the `jsp:useBean` action has a non-empty body, the body is processed. The variable is initialized and available within the scope of the body. The text of the body is treated as elsewhere. Any template text will be passed through to the out stream. Scriptlets and action tags will be evaluated.

A common use of a non-empty body is to complete initializing the created instance. In that case the

body will likely contain `jsp:setProperty` actions and scriptlets that are evaluated. This completes the processing of this `useBean` action.

Examples

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

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

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

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

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

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

Syntax

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

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

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

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

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

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

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

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

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

5.2. <jsp:setProperty>

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

There are two variants of the `jsp:setProperty` action. Both variants set the values of one or more properties in the bean based on the type of the properties. The usual bean introspection is done to

discover what properties are present, and, for each, its name, whether it is simple or indexed, its type, and the `setter` and `getter` methods. Introspection also indicates if a given property type has a `PropertyEditor` class.

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

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

When assigning from a value given as a request-time attribute, no type conversions are applied if a scripting expression is used, as indicated in Section 1.14.2.2, "Conversions from request-time expressions". If an EL expression is used, the type conversions described in Section 1.23 "Type Conversion" of the EL specification document are performed.

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

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

Examples

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

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

The following two elements set a property from a value

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

Syntax

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

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

propertyValue ::= string
```

The value `propertyValue` can also be a request-time attribute value, as described in Section 1.14.1, "Request Time Attribute Values".

```
propertyValue ::= expr_scriptlet
```

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

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

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

5.3. <jsp:getProperty>

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

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

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

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

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

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

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

Examples

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

Syntax

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

The attributes are:

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

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

5.4. <jsp:include>

A `<jsp:include .../>` action provides for the inclusion of static and dynamic resources in the same context as the current page. See **Table JSP.1-10** , "Summary of Include Mechanisms in JSP 3.0" for a summary of include facilities.

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

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

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

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

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

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

Examples

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

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

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

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

Syntax

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

and

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

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

The valid attributes are:

Table JSP.5-4 *jsp:include Atrributes*

`page`	The URL is a relative `urlSpec` as in Section 1.2.1, "Relative URL Specifications". 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`.

5.5. <jsp:forward>

A `<jsp:forward page="urlSpec" />` action allows the runtime dispatch of the current request to a static resource, a JSP page or a servlet 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 1.2.1, "Relative URL Specifications".

The request object will be adjusted according to the value of the page attribute.

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

If the page output is buffered, the buffer is cleared prior to forwarding.

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

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

Examples

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

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

Syntax

```
<jsp:forward page="relativeURLspec" />
```

and

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

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

Table JSP.5-5 *jsp:forward Attributes*

`page`	The URL is a relative `urlSpec` as in Section 1.2.1, "Relative URL Specifications". 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).

5.6. <jsp:param>

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

When doing `jsp:include` or `jsp:forward`, the included page or forwarded page will see the original request object, with the original parameters augmented with the new parameters, in the order of appearance, 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 9.1.1 in the Servlet 5.0 specification).

For example, if the request has a parameter `A=foo` and a parameter `A=bar` is specified for forward, the forwarded request shall have `A=bar,foo`. Note that the new `param` has precedence.

The parameter names and values specified should be left unencoded by the page author. The JSP container must encode the parameter names and values using the character encoding from the request object when necessary. For example, if the container chooses to append the parameters to the URL in the dispatched request, both the names and values must be encoded as per the content type `application/x-www-form-urlencoded` in the HTML specification.

Syntax

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

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

5.7. <jsp:plugin>

The plugin action enables a JSP page author to generate HTML that contains the appropriate client browser dependent constructs (`OBJECT` or `EMBED`) that will result in the download of the Java Plugin software (if required) and subsequent execution of the Applet or JavaBeans component specified therein.

The `<jsp:plugin>` tag is replaced by either an `<object>` or `<embed>` tag, as appropriate for the requesting user agent, and emitted into the output stream of the response. The attributes of the `<jsp:plugin>` tag provide configuration data for the presentation of the element, as indicated in the table below.

The `<jsp:params>` action containing one or more `<jsp:param>` actions provides parameters to the Applet or JavaBeans component.

The `<jsp:fallback>` element indicates the content to be used by the client browser if the plugin cannot be started (either because `OBJECT` or `EMBED` is not supported by the client browser or due to some other problem). If the plugin can start but the Applet or JavaBeans component cannot be found or started, a plugin specific message will be presented to the user, most likely a popup window reporting a `ClassNotFoundException`.

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

Examples

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

Syntax

```
<jsp:plugin type="bean|applet"
    code="objectCode"
    codebase="objectCodebase"
    { align="alignment"       }
    { archive="archiveList"   }
    { height="height"         }
    { hspace="hspace"         }
    { jreversion="jreversion" }
    { name="componentName"    }
    { vspace="vspace"         }
    { title="title"           }
    { width="width"           }
    { nspluginurl="url"       }
    { iepluginurl="url"       }
    { mayscript='true|false'  } >

    { <jsp:params>
        { <jsp:param name="paramName" value="paramValue" /> }+
      </jsp:params> }

    { <jsp:fallback> arbitrary_text </jsp:fallback> }
</jsp:plugin>
```

Table JSP.5-6 *jsp:plugin Attributes*

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

nspluginurl	URL where JRE plugin can be downloaded for Netscape Navigator, default is implementation defined.
iepluginurl	URL where JRE plugin can be downloaded for IE, default is implementation defined.
mayscript	As defined by HTML spec.

5.8. <jsp:params>

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

The semantics and syntax of jsp:params are described in Section 5.7, "<jsp:plugin>".

5.9. <jsp:fallback>

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

The semantics and syntax of jsp:fallback are described in Section 5.7, "<jsp:plugin>".

5.10. <jsp:attribute>

The <jsp:attribute> standard action has two uses. It allows the page author to define the value of an action attribute in the body of an XML element instead of in the value of an XML attribute. It also allows the page author to specify the attributes of the element being output, when used inside a <jsp:element> action. The action must only appear as a subelement of a standard or custom action. An attempt to use it otherwise must result in a translation error. For example, it cannot be used to specify the value of an attribute for XML elements that are template text. For custom action invocations, JSP containers must support the use of <jsp:attribute> for both Classic and Simple Tag Handlers.

The behavior of the <jsp:attribute> standard action varies depending on the type of attribute being specified, as follows:

- A translation error must occur if <jsp:attribute> is used to define the value of an attribute of <jsp:attribute>.
- If the enclosing action is <jsp:element> , the value of the name attribute and the body of the action will be used as attribute name/value pairs in the dynamically constructed element. See Section 5.14, "<jsp:element>" for more details on <jsp:element>. Note that in this context, the attribute does not apply to the <jsp:element> action itself, but rather to the output of the element. That is, <jsp:attribute> cannot be used to specify the name attribute of the <jsp:element> action.
- For custom action attributes of type jakarta.servlet.jsp.tagext.JspFragment, the container must

create a JspFragment out of the body of the <jsp:attribute> action and pass it to the tag handler. This applies for both Classic Tag Handlers and Simple Tag Handlers. A translation error must result if the body of the <jsp:attribute> action is not scriptless in this case.

- If the custom action accepts dynamic attributes (Section 7.1.8, "Attributes With Dynamic Names"), and the name of the attribute is not one explicitly indicated for the tag, then the container will evaluate the body of <jsp:attribute> and assign the computed value to the attribute using the dynamic attribute machinery. Since the type of the attribute is unknown and the body of <jsp:attribute> evaluates to a String, the container must pass in an instance of String.
- For standard or custom action attributes that accept a request-time expression value, the Container must evaluate the body of the <jsp:attribute> action and use the result of this evaluation as the value of the attribute. The body of the attribute action can be any JSP content in this case. If the type of the attribute is not String, the standard type conversion rules are applied, as per Section 1.14.2.1, "Conversions from String values".
- For standard or custom action attributes that do not accept a request-time expression value, the Container must use the body of the <jsp:attribute> action as the value of the attribute. A translation error must result if the body of the <jsp:attribute> action contains anything but template text.

If the body of the <jsp:attribute> action is empty, it is the equivalent of specifying "" as the value of the attribute. Note that after being trimmed, non-empty bodies can result in a value of "" as well.

The <jsp:attribute> action accepts a name attribute, a trim attribute, and a omit attribute. The name attribute associates the action with one of the attributes the tag handler is declared to accept, or in the case of <jsp:element> it associates the action with one of the attributes in the element being output. The optional trim attribute determines whether the whitespace appearning at the beginning and at the end of the element body should be discarded or not. By default, the leading and trailing whitespace is discarded. The Container must trim at translation time only. The Container must not trim at runtime. For example, if a body contains a custom action that produces leading or trailing whitespace, that whitespace is preserved regardless of the value of the trim attribute. The optional omit attribute, when used with <jsp:element>, determines whether the attribute in the element being output should be omitted.

The following is an example of using the <jsp:attribute> standard action to define an attribute that is evaluated by the container prior to the custom action invocation. This example assumes the name attribute is declared with type java.lang.String in the TLD.

```
<mytag:highlight>
  <jsp:attribute name="text">
      Inline definition.
  </jsp:attribute>
</mytag:highlight>
```

The following is an example of using the <jsp:attribute> standard action within <jsp:element>, to

define which attributes are to be output with that element:

```
<jsp:element name="firstname">
    <jsp:attribute name="name">Susan</jsp:attribute>
</jsp:element>
```

This would produce the following output:

```
<firstname name="Susan"/>
```

See Section 1.3.10, "JSP Syntax Grammar" for the formal syntax definition of the `<jsp:attribute>` standard action.

The attributes are:

Table JSP.5-7 *Attributes for the* `<jsp:attribute>` *standard action*

`name`	(required) If not being used with `<jsp:element>`, then if the action does not accept dynamic attributes, the name must match the name of an attribute for the action being invoked, as declared in the Tag Library Descriptor for a custom action, or as specified for a standard action, or a translation error will result. Except for when used with `<jsp:element>`, a translation error will result if both an XML element attribute and a `<jsp:attribute>` element are used to specify the value for the same attribute. The value of name can be a QName. If so, a translation error must occur if the prefix does not match that of the action it applies to, unless the action supports dynamic attributes, or unless the action is `<jsp:element>`. When used with `<jsp:element>`, this attribute specifies the name of the attribute to be included in the generated element.
`trim`	(optional) Valid values are `true` and `false`. If `true`, the whitespace, including spaces, carriage returns, line feeds, and tabs, that appears at the beginning and at the end of the body of the `<jsp:attribute>` action will be ignored by the JSP compiler. If `false` the whitespace is not ignored. Defaults to `true`.
`omit`	(optional) Valid values are `true` and `false`. If `true`, and when used with <jsp:element>, the attribute in the element being ouput is omitted. Ignored when used with a standard or custom action. Defaults to `false`.

5.11. <jsp:body>

Normally, the body of a standard or custom action invocation is defined implicitly as the body of the XML element used to represent the invocation. The body of a standard or custom action can also be defined explicitly using the `<jsp:body>` standard action. This is required if one or more `<jsp:attribute>` elements appear in the body of the tag.

If one or more <jsp:attribute> elements appear in the body of a tag invocation but no <jsp:body> element appears or an empty <jsp:body> element appears, it is the equivalent of the tag having an empty body.

It is also legal to use the <jsp:body> standard action to supply bodies to standard actions, for any standard action that accepts a body (except for <jsp:body>, <jsp:attribute>, <jsp:scriptlet>, <jsp:expression> and <jsp:declaration>).

The body standard action accepts no attributes.

5.12. <jsp:invoke>

The <jsp:invoke> standard action can only be used in tag files (see Chapter 8, *Tag Files*), and must result in a translation error if used in a JSP. It takes the name of an attribute that is a fragment, and invokes the fragment, sending the output of the result to the JspWriter, or to a scoped attribute that can be examined and manipulated. If the fragment identified by the given name is null, <jsp:invoke> will behave as though a fragment was passed in that produces no output.

5.12.1. Basic Usage

The most basic usage of this standard action will invoke a fragment with the given name with no parameters. The fragment will be invoked using the JspFragment.invoke method, passing in null for the Writer parameter so that the results will be sent to the JspWriter of the JspContext associated with the JspFragment. The following is an example of such a basic fragment invocation:

```
<jsp:invoke fragment="frag1"/>
```

5.12.2. Storing Fragment Output

It is also possible to invoke the fragment and send the results to a scoped attribute for further examination and manipulation. This can be accomplished by specifying the var or varReader attribute in the action. In this usage, the fragment is invoked using the JspFragment.invoke method, but a custom java.io.Writer is passed in instead of null.

If var is specified, the container must ensure that a java.lang.String object is made available in a scoped attribute with the name specified by var. The String must contain the content sent by the fragment to the Writer provided in the JspFragment.invoke call.

If varReader is specified, the container must ensure that a java.io.Reader object is constructed and is made available in a scoped attribute with the name specified by varReader. The Reader object can then be passed to a custom action for further processing. The Reader object must produce the content sent by the fragment to the provided Writer. The Reader must also be resettable. That is, if its reset method is called, the result of the invoked fragment must be able to be read again without re-executing the fragment.

An optional `scope` attribute indicates the scope of the resulting scoped variable.

The following is an example of using `var` or `varReader` and the `scope` attribute:

```
<jsp:invoke fragment="frag2" var="resultString" scope="session"/>
```

```
<jsp:invoke fragment="frag3" varReader="resultReader" scope="page"/>
```

5.12.3. Providing a Fragment Access to Variables

JSP fragments have access to the same page scope variables as the page or tag file in which they were defined (in addition to variables in the request, session, and application scopes). Tag files have access to a local page scope, separate from the page scope of the calling page. When a tag file invokes a fragment that appears in the calling page, the JSP container provides a way to synchronize variables between the local page scope in the tag file and the page scope of the calling page. For each variable that is to be synchronized, the tag file author must declare the variable with a scope of either `AT_BEGIN` or `NESTED`. The container must then generate code to synchronize the page scope values for the variable in the tag file with the page scope equivalent in the calling page or tag file. The details of how variables are synchronized can be found in Section 8.9, "Variable Synchronization".

The following is an example of a tag file providing a fragment access to a variable:

```
<%@ variable name-given="x" scope="NESTED" %>
...
<c:set var="x" value="1"/>
<jsp:invoke fragment="frag4"/>
```

A translation error shall result if the `<jsp:invoke>` action contains a non-empty body.

See Section 1.3.10, "JSP Syntax Grammar" for the formal syntax definition of the `<jsp:invoke>` standard action.

The attributes are:

Table JSP.5-8 *Attributes for the* `<jsp:invoke>` *standard action*

`fragment`	(required) The name used to identify this fragment during this tag invocation.
`var`	(optional) The name of a scoped attribute to store the result of the fragment invocation in, as a `java.lang.String` object. A translation error must occur if both `var` and `varReader` are specified. If neither `var` nor `varReader` are specified, the result of the fragment goes directly to the `JspWriter`, as described above.

varReader	(optional) The name of a scoped attribute to store the result of the fragment invocation in, as a java.io.Reader object. A translation error must occur if both var and varReader are specified. If neither var nor varReader is specified, the result of the fragment invocation goes directly to the JspWriter, as described above.
scope	(optional) The scope in which to store the resulting variable. A translation error must result if the value is not one of page, request, session, or application. A translation error will result if this attribute appears without specifying either the var or varReader attribute as well. Note that a value of session should be used with caution since not all calling pages may be participating in a session. A container must throw an IllegalStateException at runtime if scope is session and the calling page does not participate in a session. Defaults to page.

5.13. <jsp:doBody>

The <jsp:doBody> standard action can only be used in tag files (see Chapter 8, *Tag Files*), and must result in a translation error if used in a JSP. It invokes the body of the tag, sending the output of the result to the JspWriter, or to a scoped attribute that can be examined and manipulated.

The <jsp:doBody> standard action behaves exactly like <jsp:invoke>, except that it operates on the body of the tag instead of on a specific fragment passed as an attribute. Because it always operates on the body of the tag, there is no name attribute for this standard action. The var, varReader, and scope attributes are all supported with the same semantics as for <jsp:invoke>. Fragments are provided access to variables the same way for <jsp:doBody> as they are for <jsp:invoke>. If no body was passed to the tag, <jsp:doBody> will behave as though a body was passed in that produces no output.

The body of a tag is passed to the simple tag handler as a JspFragment object.

A translation error shall result if the <jsp:doBody> action contains a non-empty body.

See Section 1.3.10, "JSP Syntax Grammar" for the formal syntax definition of the <jsp:doBody> standard action.

The attributes are:

Table JSP.5-9 *Attributes for the* <jsp:doBody> *standard action*

var	(optional) The name of a scoped attribute to store the result of the body invocation in, as a java.lang.String object. A translation error must occur if both var and varReader are specified. If neither var nor varReader are specified, the result of the body goes directly to the JspWriter, as described above.
varReader	(optional) The name of a scoped attribute to store the result of the body invocation in, as a java.io.Reader object. A translation error must occur if both var and varReader are specified. If neither var nor varReader is specified, the result of the body invocation goes directly to the JspWriter, as described above.

`scope`	(optional) The scope in which to store the resulting variable. A translation error must result if the value is not one of `page`, `request`, `session`, or `application`. A translation error will result if this attribute appears without specifying either the `var` or `varReader` attribute as well. Note that a value of `session` should be used with caution since not all calling pages may be participating in a session. A container must throw an `IllegalStateException` at runtime if `scope` is `session` and the calling page does not participate in a session. Defaults to `page`.

5.14. <jsp:element>

The `jsp:element` action is used to dynamically define the value of the tag of an XML element. This action can be used in JSP pages, tag files and JSP documents. This action has an optional body; the body can use the `jsp:attribute and jsp:body actions.`

A `jsp:element` action has one mandatory attribute, name, of type `String`. The value of the attribute is used as that of the tag of the element generated.

Examples

The following example generates an XML element whose name depends on the result of an EL expression, content.headerName. The element has an attribute, lang, and the value of the attribute is that of the expression content.lang. The body of the element is the value of the expression content.body.

```
<jsp:element
      name="${content.headerName}"
      xmlns:jsp="http://java.sun.com/JSP/Page" >
  <jsp:attribute name="lang">${content.lang}</jsp:attribute>
  <jsp:body>${content.body}</jsp:body>
</jsp:element>
```

The next example fragment shows that `jsp:element` needs no children. The example generates an empty element with name that of the value of the expression myName.

```
<jsp:element name="${myName}"/>
```

Syntax

The `jsp:element` action may have a body. Two forms are valid, depending on whether the element is to have attributes or not. In the first form, no attributes are present:

```
<jsp:element name="name">
  optional body
</jsp:element>
```

In the second form, zero or more attributes are requested, using `jsp:attribute` and `jsp:body`, as appropriate.

```
<jsp:element name="name">
  jsp:attribute*
  jsp:body?
</jsp:element>
```

The one valid, mandatory, attribute of `jsp:element` is its name. Unlike other standard actions, the value of the `name` attribute must be given as an XML-style attribute and cannot be specified using `<jsp:attribute>` This is because `<jsp:attribute>` has a special meaning when used in the body of `<jsp:element>`. See Section 5.10, "<jsp:attribute>" for more details..

Table JSP.5-10 *Attributes for the `<jsp:element>` standard action*

`name`	(required) The value of name is that of the element generated. The name can be a QName; JSP 3.0 places no constraints on this value: it is accepted as is. A request-time attribute value may be used for this attribute.

5.15. <jsp:text>

A `jsp:text` action can be used to enclose template data in a JSP page, a JSP document, or a tag file. A `jsp:text` action has no attributes and can appear anywhere that template data can. Its syntax is:

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

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

Examples

The following example is a fragment that could be in both a JSP page or a JSP document.

```
<jsp:text>
  This is some content
</jsp:text>
```

Expressions may appear within `jsp:text`, as in the next example, where the expression *foo.content* is

evaluated and the result is inserted.

```
<jsp:text>
  This is some content: ${foo.content}
</jsp:text>
```

No subelements may appear within `jsp:text` ; for example the following fragment is invalid and must generate a translation error.

```
<jsp:text>
 This is some content: <jsp:text>foo</jsp:text>
</jsp:text>
```

When within a JSP document, of course, the body content needs to additionally conform to the constraints of being a well-formed XML document, so the following example, although valid in a JSP page is invalid in a JSP document:

```
<jsp:text>
  This is some content: ${foo.content > 3}
</jsp:text>
```

The same example can be made legal, with no semantic changes, by using `gt` instead of `>` in the expression; i.e. `${foo.content gt 3}`.

In an JSP document, CDATA sections can also be used to quote, uninterpreted, content, as in the following example:

```
<jsp:text>
  <![CDATA[<mumble></foobar>]]>
</jsp:text>
```

Syntax

The `jsp:text` action has no attributes. The action may have a body. The body may not have nested actions nor scripting elements. The body may have EL expressions. The syntax is of the form:

```
<jsp:text>
  optional body
</jsp:text>
```

5.16. <jsp:output>

The `jsp:output` action can only be used in JSP documents and in tag files in XML syntax, and a translation error must result if used in a standard syntax JSP or tag file. This action is used to modify some properties of the output of a JSP document or a tag file. In JSP 3.0 there are four properties that can be specified, all of which affect the output of the XML prolog.

The `omit-xml-declaration` property allows the page author to adjust whether an XML declaration is to be inserted at the beginning of the output. Since XML declarations only make sense for when the generated content is XML, the default value of this property is defined so that it is unnecessary in most cases.

The `omit-xml-declaration` property is of type `String` and the valid values are "yes", "no", "true" and "false". The name, values and semantics mimic that of the `xsl:output` element in the XSLT specification: if a value of "yes" or "true" is given, the container will not add an XML declaration; if a value of "no" or "false" is given, the container will add an XML declaration.

The default value for a JSP document that has a `jsp:root` element is "yes". The default value for JSP documents without a `jsp:root` element is "no".

The default value for a tag file in XML syntax is always "yes". If the value is "false" or "no" the tag file will emit an XML declaration as its first content.

The generated XML declaration is of the form:

```
<?xml version="1.0" encoding="encodingValue" ?>
```

Where `encodingValue` is the response character encoding, as determined in Section 4.2, “Response Character Encoding”.

The `doctype-root-element`, `doctype-system` and `doctype-public` properties allow the page author to specify that a DOCTYPE be automatically generated in the XML prolog of the output. Without these properties, the DOCTYPE would need to be output manually via a `<jsp:text>` element before the root element of the JSP document, which is inconvenient.

A DOCTYPE must be automatically output if and only if the `doctype-system` element appears in the translation unit as part of a `<jsp:output>` action. The `doctype-root-element` must appear and must only appear if the `doctype-system` property appears, or a translation error must occur. The `doctype-public` property is optional, but must not appear unless the `doctype-system` property appears, or a translation error must occur.

The DOCTYPE to be automatically output, if any, is statically determined at translation time. Multiple occurrences of the `doctype-root-element`, `doctype-system` or `doctype-public` properties will cause a translation error if the values for the properties differ from the previous occurrence.

The DOCTYPE that is automatically output, if any, must appear immediately before the first element of the output document. The name following <!DOCTYPE must be the value of the `doctype-root-element` property. If a `doctype-public` property appears, then the format of the generated DOCTYPE is:

```
<!DOCTYPE nameOfRootElement PUBLIC "doctypePublic" "doctypeSystem">
```

If a `doctype-public` property does not appear, then the format of the generated DOCTYPE is:

```
<!DOCTYPE nameOfRootElement SYSTEM "doctypeSystem">
```

Where `nameOfRootElement` is the value of the `doctype-root-element` property, `doctypePublic` is the value of the `doctype-public` attribute, and `doctypeSystem` is the value of the `doctype-system` property.

The values for `doctypePublic` and `doctypeSystem` must be enclosed in either single or double quotes, depending on the value provided by the page author. It is the responsibility of the page author to provide a syntactically-valid URI as per the XML specification (see http://www.w3.org/TR/REC-xml#dt-sysid).

Examples

The following JSP document (with an extension of `.jspx` or with `<is-xml>` set to `true` in the JSP configuration):

```
<?xml version="1.0" encoding="EUC-JP" ?>
<hello></hello>
```

generates an XML document as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<hello></hello>
```

The following JSP document is like the previous one, except that the XML declaration is omited. A typical use would be where the XML fragment is to be included within another document.

```
<?xml version="1.0" encoding="EUC-JP" ?>
<hello>
  <jsp:output
      xmlns:jsp="http://java.sun.com/JSP/Page"
      omit-xml-declaration="true"/>
</hello>
```

The following JSP document is equivalent but uses `jsp:root` instead of `jsp:output`.

```
<?xml version="1.0" encoding="EUC-JP" ?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0" >
  <hello></hello>
</jsp:root>
```

The following JSP document specifies both a `doctype-public` and a `doctype-system` :

```
<?xml version="1.0" encoding="UTF-8" ?>
<html xmlns:jsp="http://java.sun.com/JSP/Page">
  <jsp:output doctype-root-element="html"
      doctype-public="-//W3C//DTD XHTML Basic 1.0//EN"
      doctype-system="http://www.w3.org/TR/xhtml-basic/xhtml-basic10.dtd" />
  <body>
    <h1>Example XHTML Document</h1>
  </body>
</html>
```

and generates and XML document as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML Basic 1.0//EN"
"http://www.w3.org/TR/xhtml-basic/xhtml-basic10.dtd">
<html><body><h1>Example XHTML Document</h1></body></html>
```

The following JSP document omits the `doctype-public` and explicitly omits the XML declaration:

```
<?xml version="1.0" encoding="UTF-8" ?>
<elementA>
  <jsp:output omit-xml-declaration="true"
      doctype-root-element="elementA"
      doctype-system="test.dtd" />
  Element body goes here.
</elementA>
```

and generates an XML document as follows:

```
<!DOCTYPE elementA SYSTEM "test.dtd">
<elementA>Element body goes here.</elementA>
```

Syntax

The jsp:output action cannot have a body. The <jsp:output> action has the following syntax:

```
<jsp:output ( omit-xml-declaration="yes|no|true|false" ) { doctypeDecl } />

doctypeDecl ::=   ( doctype-root-element="rootElement"
                    doctype-public="PubidLiteral"
                    doctype-system="SystemLiteral" )
                | ( doctype-root-element="rootElement"
                    doctype-system="SystemLiteral" )
```

The following are the valid attributes of jsp:output:

Table JSP.5-11 *Attribute for the* <jsp:output> *standard action*

omit-xml-declaration	(optional) Indicates whether to omit the generation of an XML declaration. Acceptable values are "true", "yes", "false" and "no".
doctype-root-element	(optional) Must be specified if and only if doctype-system is specified or a translation error must occur. Indicates the name that is to be output in the generated DOCTYPE declaration.
doctype-system	(optional) Specifies that a DOCTYPE declaration is to be generated and gives the value for the System Literal.
doctype-public	(optional) Must not be specified unless doctype-system is specified. Gives the value for the Public ID for the generated DOCTYPE.

5.17. Other Standard Actions

Chapter 6, *JSP Documents* defines several other standard actions that are either convenient or needed to describe JSP pages with an XML document, some of which are available in all JSP pages. They are:

- <jsp:root>, defined in Section 6.3.2, “The jsp:root Element”.
- <jsp:declaration>, defined in Section 6.3.7, “Scripting Elements”.
- <jsp:scriptlet>, defined in Section 6.3.7, “Scripting Elements”.
- <jsp:expression>, defined in Section 6.3.7, “Scripting Elements”.

Chapter 6. JSP Documents

This chapter introduces two concepts related to XML and JSP: JSP documents and XML views. This chapter provides a brief overview of the two concepts and their relationship and also provides the details of JSP documents. The details of the XML view of a JSP document are described in Chapter 10, *XML View*.

6.1. Overview of JSP Documents and of XML Views

A *JSP document* is a JSP page written using XML syntax. JSP documents need to be described as such, either implicitly or explicitly, to the JSP container, which will then process them as XML documents, checking for well-formedness and applying requests like entity declarations, if present. JSP documents are used to generate dynamic content using the standard JSP semantics.

Here is a simple JSP document:

```
<table>
  <c:forEach
      xmlns:c="http://java.sun.com/jsp/jstl/core"
      var="counter" begin="1" end="3">
    <row>${counter}</row>
  </c:forEach>
</table>
```

This well-formed, namespace-aware XML document generates, using the JSP standard tag library, an XML document that has `<table>` as the root element. That element has 3 `<row>` subelements containing values `1`, `2` and `3`. See Section 6.4, “Examples of JSP Documents” for more details of this and other examples.

The design of JSP documents is focused on the generation of dynamic XML content, in any of its many uses, but JSP documents can be used to generate any dynamic content.

Some of the syntactic elements described in Chapter 1, *Core Syntax and Semantics* are not legal XML; this chapter describes alternative syntaxes for those elements that are aligned with the XML syntax. The alternative syntaxes can be used in JSP documents; most of them (`jsp:output` and `jsp:root` are exceptions) can also be used in JSP pages in JSP syntax. As it will be described later, the alternative syntax is also used in the XML view of JSP pages.

JSP documents can be used in a number of ways, including:

- JSP documents can be passed directly to the JSP container; this is becoming more important as more and more content is authored as XML, be it in an XML-based languages like XHTML or SVG, or for the exchange of documents in applications like Web Services. The generated content may be sent directly to a client, or it may be part of some XML processing pipeline.

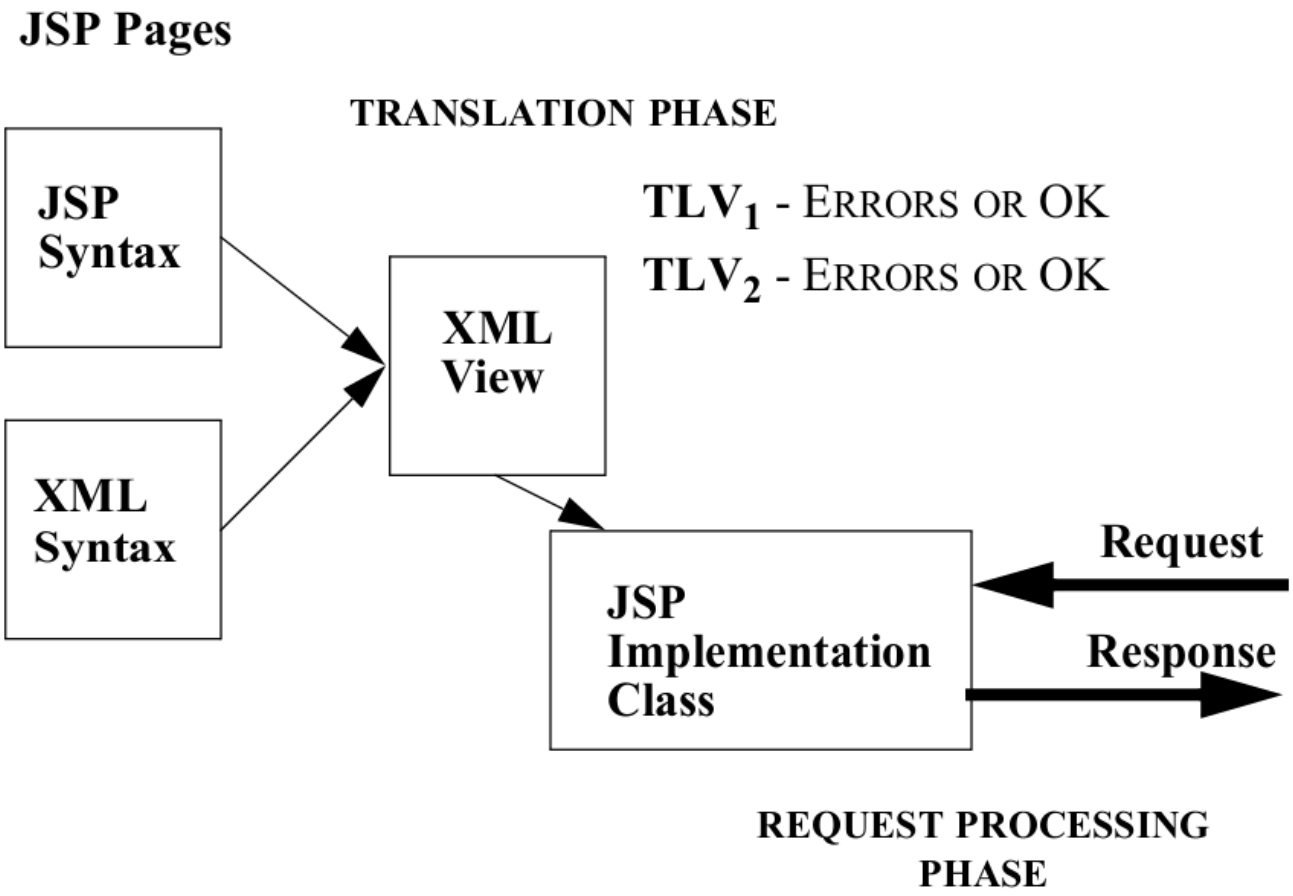
- JSP documents can be manipulated by XML-aware tools.
- A JSP document can be generated from a textual representation by applying an XML transformation, like XSLT.
- A JSP document can be generated automatically, say by serializing some objects.

Tag files can also be authored using XML syntax. The rules are very similar to that of JSP documents; see Section 8.6, "Tag Files in XML Syntax" for more details.

The *XML view of a JSP page* is an XML document that is derived from the JSP page following a mapping defined later in this chapter. The XML view of a JSP page is intended to be used for validating the JSP page against some description of the set of valid pages. Validation of the JSP page is supported in the JSP 3.0 specification through a `TagLibraryValidator` class associated with a tag library. The validator class acts on a PageData object that represents the XML view of the JSP page (see, for example, Section 7.4.1.2, "Validator Classes")

Figure JSP.6-1 Relationship between JSP Pages and XML views of JSP pages below depicts the relationship between the concepts of JSP pages, JSP documents and XML views. Two phases are involved: the Translation phase, where JSP pages, in either syntax, are exposed to Tag Library Validators, via their XML view, and the Request Processing phase, where requests are processed to produce responses.

Figure JSP.6-1 *Relationship between JSP Pages and XML views of JSP pages*



JSP documents are used by JSP page authors. They can be authored directly, using a text editor, through an XML editing tool, or through a JSP page authoring tool that is aware of the XML syntax. Any JSP page author that is generating XML content should consider the use of JSP documents. In contrast, the XML view of a JSP page is a concept internal to the JSP container and is of interest only to Tag Library Authors and to implementors of JSP containers.

6.2. JSP Documents

A JSP document is a JSP page that is a namespace-aware XML document and that is identified as a JSP document to the JSP container.

6.2.1. Identifying JSP Documents

A JSP document can be identified as such in three ways:

- If there is a `<jsp-property-group>` that explicitly indicates, through the `<is-xml>` element, whether a given file is a JSP document, then that indication overrides any other determination. Otherwise,
- If this web application is using a version 2.4 web.xml, and if the extension is . `jspx`, then the file is a JSP document. Otherwise,
- If the file is explicitly or implicitly identified as a JSP page and the top element is a `jsp:root` element then the file is identified as a JSP document. This behavior provides backwards compatibility with JSP 1.2.

It is a translation-time error for a file that is identified as a JSP document to not be a well-formed, namespace-aware, XML document.

See Section 8.6, “Tag Files in XML Syntax” for details on identifying tag files in XML syntax.

6.2.2. Overview of Syntax of JSP Documents

A JSP document may or not have a `<jsp:root>` as its top element; `<jsp:root>` was mandatory in JSP 1.2, but we expect most JSP documents in JSP 3.0 not to use it.

JSP documents identify standard actions through the use of a well-defined URI in its namespace; although in this chapter the prefix `jsp` is used for the standard actions, any prefix is valid as long as the correct URI identifying JSP 3.0 standard actions is used. Custom actions are identified using the URI that identifies their tag library; `taglib` directives are not required and cannot appear in a JSP document.

A JSP document can use XML elements as template data; these elements may have qualified names (and thus be in a namespace), or be unqualified.

The `<jsp:text>` element can be used to define some template data verbatim.

Since a JSP document must be a valid XML document, there are some JSP elements that can’t be used in

a JSP document. The elements that can be used are:

- JSP directives and scripting elements in XML syntax.
- EL expressions in the body of elements and in attribute values.
- All JSP standard actions described in Chapter 1, *Core Syntax and Semantics*.
- The jsp:root, jsp:text, and jsp:output elements.
- Custom action elements.
- Template data described using `jsp:text` elements.
- Template data described through XML fragments.

Scriptlet expressions used to specify request-time attribute values use a slightly different syntax in JSP documents than in non JSP documents; rather than using `<%= expr %>`, they use `%= expr %`. The white space around `expr` is not needed, and note the missing `<` and `>`. The `expr`, after any applicable quoting as in any other XML document, is an expression to be evaluated as in Section 1.14.1, "Request Time Attribute Values".

The mechanisms that enable scripting and EL evaluation in a JSP page apply also when the page is a JSP document. Just as in the standard syntax, the `$` in an EL expression can be quoted using `\$` in both attribute values and template text. Recall, however, that `\\` is not an escape sequence in XML attributes so whereas within an attribute in standard syntax `\\${1+1}` would result in `\2` (assuming EL is enabled) or `\${1+1}` (assuming EL is ignored), in XML syntax `\\${1+1}` always results in `\${1+1}`.

It should be noted that the equivalent JSP document form of
`<a href="<%= url %>">`, where 'a' is not a custom action, is:

```
<jsp:text><![CDATA[<a href="]]></jsp:text><jsp:expression>url</jsp:expression>
<jsp:text><![CDATA[">]]></jsp:text>
```

In the JSP document element `<a href="%= url %">`, `"%= url %"` does not represent a request-time attribute value. That syntax only applies for custom action elements. This is in contrast to `<a href="${url}">`, where "${url}" represents an EL expression in both JSP pages and JSP documents.

6.2.3. Semantic Model

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

The first step in processing a JSP document is to process it as an XML document, checking for well-formedness, processing entity resolution and, if applicable, performing validation as described in

Section 6.2.4, “JSP Document Validation”. As part of the processing XML quoting will be performed, and JSP quoting will not be performed later.

After these steps, the JSP document will be passed to the JSP container which will then interpret it as a JSP page.

The JSP processing step for a JSP document is as for any other JSP page except that namespaces are used to identify standard actions and custom action tag libraries and that run time expressions in attributes use the slightly different syntax. Note that all the JSP elements that are described in this chapter are valid in all JSP pages, be they identified as JSP documents or not. This is a backward compatible change from the behavior in JSP 1.2 to enable gradual introduction of XML syntax in existing JSP pages.

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

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

The container will add, in some conditions, an XML declaration to the output; the rules for this depend on the use of jsp:root and jsp:output; see Section 6.3.3, “The jsp:output Element”.

6.2.4. JSP Document Validation

A JSP document with a DOCTYPE declaration must be validated by the container in the translation phase. Validation errors must be handled the same way as any other translation phase errors, as described in Section 1.4.1, “Translation Time Processing Errors”.

JSP 3.0 requires only DTD validation for JSP Documents; containers should not perform validation based on other types of schemas, such as XML schema.

If an author wishes to have the JSP document framed by the root element of a vocabulary outside the http://java.sun.com/JSP/Page namespace, and they wish to be able to validate the JSP document according to a DTD, then they should be aware that the DTD must make explicit provision for elements from the JSP namespace, and the namespace prefix to which they are bound.

For example, the following XML document:

```
<?xml version="1.0"?>
<!DOCTYPE root PUBLIC "-//My Org//My DTD//EN"
    "http://www.my.org/dtd/my.dtd">
<root xmlns:jsp="http://java.sun.com/JSP/Page"/>
```

can only be validated against its DTD if the DTD makes special provision for both the attribute "xmlns:jsp" on the root element, and also for elements with a "jsp" namespace prefix. Even if the DTD provides for this, you must bind the namespace to the prefix that the DTD has chosen.

6.3. Syntactic Elements in JSP Documents

This section describes the elements in a JSP document.

6.3.1. Namespaces, Standard Actions, and Tag Libraries

JSP documents and tag files in XML syntax use XML namespaces to identify the standard actions, the directives, and the custom actions. JSP pages and tags in the JSP syntax cannot use XML namespaces and instead must use the taglib directive.

Though the prefix “jsp” is used throughout this specification, it is the namespace `http://java.sun.com/JSP/Page` and not the prefix “jsp” that identifies the JSP standard actions.

An `xmlns` attribute for a custom tag library of the form *xml:prefix='uri'* identifies the tag library through the `uri` value. The `uri` value may be of one of three forms, either a URN of the form `urn:jsptagdir:tagdir`, a URN of the form `urn:jsptld:path`, or a plain URI.

If the `uri` value is a URN of the form `urn:jsptld:path`, then the TLD is determined following the mechanism described in Section 7.3.2, “TLD Resource Path”.

If the `uri` value is a URN of the form `urn:jsptagdir:tagdir`, then the TLD is determined following the mechanism described in Section 8.4, “Packaging Tag Files”.

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 Section 7.3.3, “Taglib Map in web.xml” and Section 7.3.4, “Implicit Map Entries from TLDs”), as indicated in Section 7.3.6, “Determining the TLD Resource Path”. In contrast to Section 7.3.6.2, “Computing the TLD Resource Path”, however, a translation error must not be generated if the given `uri` is not found in the taglib map. Instead, any actions in the namespace defined by the uri value must be treated as uninterpreted.

6.3.2. The jsp:root Element

The `jsp:root` element can only appear as the root element in a JSP document or in a tag file in XML syntax; otherwise a translation error shall occur. JSP documents and tag files in XML syntax need not have a `jsp:root` element as its root element.

The `jsp:root` element has two main uses. One is to indicate that the JSP file is in XML syntax, without having to use configuration group elements nor using the .jspx extension. The other use of the `jsp:root` element is to accomodate the generation of content that is not a single XML document: either a sequence of XML documents or some non-XML content.

A `jsp:root` element can be used to provide zero or more `xmlns` attributes that correspond to

namespaces for the standard actions, for custom actions or for generated template text. Unlike in JSP 1.2, not all tag libraries used within the JSP document need to be introduced on the root; tag libraries can be incorporated as needed inside the document using additional `xmlns` attributes.

The jsp:root element has one mandatory element, the version of the JSP spec that the page is using.

When `jsp:root` is used, the container will, by default, not insert an XML declaration; the default can be changed using the `jsp:output` element.

Examples

The following example generates a sequence of two XML documents. No XML declaration is generated.

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
  <table>foo</table>
  <table>bar</table>
</jsp:root>
```

The following example generates one XML document. An XML declaration is generated because of the use of `jsp:output`. The example is mostly instructional, as the same content could be generated dropping the `jsp:root` element.

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
  <jsp:output omit-xml-declaration="no"/>
  <table>foo</table>
</jsp:root>
```

Syntax

No other attributes are defined in this element.

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" body...
</jsp:root>
```

The one valid, mandatory, attribute of jsp:root is the version of the JSP specification used:

Table JSP.6-1 *Attributes for the* `<jsp:root>` *standard action*

`version`	(required) The version of the JSP specification used in this page. Valid values are "1.2", "2.0", and "2.1". It is a translation error if the container does not support the specified version.

6.3.3. The jsp:output Element

The jsp:output element can be used in JSP documents and in tag files in XML syntax. The jsp:output element is described in detail in Section 5.16, "<jsp:output>".

6.3.4. The jsp:directive.page Element

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

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

Where `page_directive_attr_list` is as described in Section 1.10.1, "The `page` Directive".

The interpretation of a `jsp:directive.page` element is as described in Section 1.10.1, "The `page` Directive", and its scope is the JSP document and any fragments included through an include directive.

6.3.5. The jsp:directive.include Element

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

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

The interpretation of a `jsp:directive.include` element is as in Section 1.10.3, "The `include` Directive".

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.

6.3.6. Additional Directive Elements in Tag Files

Chapter 8, *Tag Files* describes the tag, attribute and variable directives, which can be used in tag files. The XML syntax for these directives is the same as in the XML view (see Section 10.1.14, "The tag Directive", Section 10.1.15, "The attribute Directive", and Section 10.1.16, "The variable Directive" for details).

6.3.7. Scripting Elements

The usual scripting elements: declarations, scriptlets and expressions, can be used in JSP documents, but the only valid forms for these elements in a JSP document are the XML syntaxes; i.e. those using the elements `jsp:declaration`, `jsp:scriptlet` and `jsp:expression`.

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

itself. The interpretation of a `jsp:declaration` element is as in Section 1.12.1, “Declarations”. Its syntax is:

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

The `jsp:scriptlet` element is used to describe actions to be performed in response to some request. Scriptlets are program fragments. A `jsp:scriptlet` element has no attributes and its body is the program fragment that comprises the scriptlet. The interpretation of a `jsp:scriptlet` element is as in Section 1.12.2, “Scriptlets”. Its syntax is:

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

The `jsp:expression` element is used to describe complete expressions in the scripting language that get evaluated at response time. A `jsp:expression` element has no attributes and its body is the expression. The interpretation of a `jsp:expression` element is as in Section 1.12.3, “Expressions”. Its syntax is:

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

6.3.8. Other Standard Actions

The standard actions of Chapter 5, *Standard Actions* use a syntax that is consistent with XML syntax and they can be used in JSP documents and in tag files in XML syntax.

6.3.9. Template Content

A JSP page has no structure on its template content, and, correspondingly, imposes no constraints on that content. On the other hand, JSP documents have structure and some constraints are needed.

JSP documents can generate unconstrained content using `jsp:text`, as defined in Section 5.15, “<jsp:text>”. `jsp:text` can be used to generate totally fixed content but it can also be used to generate some dynamic content, as described in Section 6.3.10, “Dynamic Template Content” below.

Fixed structured content can be generated using XML fragments. A template XML element, an element that represents neither a standard action nor a custom action, can appear anywhere where a `jsp:text` may appear in a JSP document. The interpretation of such an XML element is to pass its textual representation to the current value of `out`, after the whitespace processing described in Section 6.2.3, “Semantic Model”.

For example, if the variable i has the value 3, and the JSP document is of the form. :

Table JSP.6-2 *Example 1 - Input*

LineNo	Source Text
1	<hello>
2	<hi>
3	<jsp:text> hi you all
4	</jsp:text>${i}
5	</hi>
6	</hello>

The result is:

Table JSP.6-3 *Example 1 - Output*

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

6.3.10. Dynamic Template Content

Custom actions can be used to generate any content, both structured and unstructured. Future versions of the JSP specification may allow for custom actions to check constraints on the generated content (see Section 6.5.1, “Generating XML Content Natively”) but the current specification has no standards support for any such constraints.

The most flexible standard mechanism for dynamic content is `jsp:element`. `jsp:element`, together with `jsp:attribute` and `jsp:body` can be used to generate any element. Further details of jsp:element, jsp:attribute and jsp:body are given in Section 5.14, “<jsp:element>”, in Section 5.10, “<jsp:attribute>” and in Section 5.11, “<jsp:body>”. The following example is from that section:

```
<jsp:element
    name="${content.headerName}"
    xmlns:jsp="http://java.sun.com/JSP/Page" >
  <jsp:attribute name="lang">${content.lang}</jsp:attribute>
  <jsp:body>${content.body}</jsp:body>
</jsp:element>
```

In some cases, the dynamic content that is generated can be described as simple substitutions on otherwise static templates. JSP documents can have XML templates where EL expressions are used as the values of the body or of attributes. For instance, the next example uses the expression `table.indent` as the value of an attribute, and the expression `table.value` as that for the body of an element:

```
<table indent="${table.indent}">
  <row>${table.value}</row>
</table>
```

6.4. Examples of JSP Documents

The following sections provide several annotated examples of JSP documents.

6.4.1. Example: A Simple JSP Document

This simple JSP document generates a table with 3 rows with numeric values 1, 2, 3. The JSP document uses template XML elements intermixed with actions from the JSP Standard Tag Library.

```
<table size="${3}">
  <c:forEach
      xmlns:c="http://java.sun.com/jsp/jstl/core"
      var="counter" begin="1" end="${3}">
    <row>${counter}</row>
  </c:forEach>
</table>
```

Some comments:

- The XML template elements are <table> and <row>. The custom action element is <c:forEach>
- The JSP standard tag library is introduced through the use of its URI namespace and the specific prefix used, c in this case, is irrelevant. The prefix is introduced in a non-root element, and the top element of the document is still <table>.
- The expression ${counter} is used within the <row> template element.
- The expression ${3} (3 would have been equally good, but an expression is used for expository reasons) is used within the value of an attribute in both the XML template element <table> and in the custom action element <c:forEach>.
- The JSP document does not have an xml declaration - we are assuming the encoding of the file did not require it, e.g. it used UTF-8, - but the output will include an xml declaration due to the defaulting rules and to the absence of jsp:output element directing the container to do otherwise.

The JSP document above does not generate an XML document that uses namespaces, but the next example does.

6.4.2. Example: Generating Namespace-aware Documents

```
<table
    xmlns="http://table.com/Table1"
    size="${3}">
  <c:forEach
      xmlns:c="http://java.sun.com/jsp/jstl/core"
      var="counter" begin="1" end="${3}">
    <row>${counter}</row>
  </c:forEach>
</table>
```

This example is essentially the same as the one above, except that a default namespace is introduced in the top element The namespace applies to the unqualified elements: <table> and <row>. Also note that if the default namespace were to correspond to a custom action, then the elements so effected would be interpreted as invocations on custom actions or tags.

Although the JSP container understands that this document is a namespace-aware document, the JSP 3.0 container does not really understand that the generated content is a well-formed XML document and, as the next example shows, a JSP document can generate other types of content.

6.4.3. Example: Generating non-XML documents

```
<jsp:root
    xmlns:c="http://java.sun.com/jsp/jstl/core"
    xmlns:jsp="http://java.sun.com/JSP/Page"
    version="2.0">
  <c:forEach
      var="counter" begin="1" end="${3}">
    <jsp:text>${counter}</jsp:text>
  </c:forEach>
</jsp:root>
```

This example just generates 123. There is no xml declaration generated because there is no <jsp:output> element to modify the default rule for when a JSP document has <jsp:root>. No additional whitespace is introduced because there is none within the <jsp:text> element.

The previous example used elements in the JSP namespace. That example used the jsp prefix, but, unlike with JSP pages in JSP syntax, the name of the prefix is irrelevant (although highly convenient) in JSP documents: the JSP URI is the only important indicative and the corrent URI should be used, and introduced via a namespace attribute.

For example, the same output would be generated with the following modification of the previous example:

```
<wombat:root
    xmlns:c="http://java.sun.com/jsp/jstl/core"
    xmlns:wombat="http://java.sun.com/JSP/Page"
    version="2.0">
  <c:forEach
      var="counter" begin="1" end="${3}">
    <wombat:text>${counter}</wombat:text>
  </c:forEach>
</wombat:root>
```

On the other hand, although the following example uses the jsp prefix the URI used in the namespace attribute is not the JSP URI and the JSP document will generate as output an XML document with root <jsp:root> using the URI http://johnsonshippingproducts.com.

```
<jsp:root
    xmlns:c="http://java.sun.com/jsp/jstl/core"
    xmlns:jsp="http://johnsonshippingproducts.com"
    version="2.0">
  <c:forEach
      var="counter" begin="1" end="${3}">
    <jsp:text>${counter}</jsp:text>
  </c:forEach>
</jsp:root>
```

Finally, note that, since a JSP document is a well-formed, namespace-aware document, prefixes, including jsp cannot be used without being introduced through a namespace attribute.

6.4.4. Example: Using Custom Actions and Tag Files

Custom actions are frequently used within a JSP document to generate portions of XML content. The JSP specification treats this content as plain text, with no intepretation nor constraints imposed on it. Good practice, though, suggests abstractions that organize the content along well-formed fragments.

The following example generates an XHTML document using tag library abstractions for presentation and data access, made available through the prefixes u and data respectively.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:jsp="http://java.sun.com/JSP/Page"
      xmlns:c="http://java.sun.com/jsp/jstl/core"
      xmlns:u="urn:jsptagdir:/WEB-INF/tags/mylib/"
      xmlns:data="http://acme.com/functions">
  <c:set var="title" value="Best Movies" />
  <u:headInfo title="${title}"/>
  <body>
    <h1>${title}</h1>
    <h2>List of Best Movies</h2>
    <ul>
      <c:forEach var="m" varStatus="s" items="data:movieItems()">
        <li><a href="#EL${s.index}">${s.index}</a>${m.title}</li>
      </c:forEach>
    </ul>
  </body>
</html>
```

For convenience we use the <c:set> JSTL action, which defines variables and associates values with them. This allows grouping in a single place of definitions used elsewhere.

Notice that if the above example included a DOCTYPE declaration for XHTML documents, it would not validate according to the DTD for XHTML documents, because that DTD does not list any of the namespaces declared on the <html> root element as valid attributes on the <html> element type.

However, to output a DOCTYPE, the <jsp:output> standard action specified in Section 5.16, “<jsp:output>” could be used.

The action <u:headInfo> could be implemented either through a custom action or through a tag. For example, as a tag it could be defined by the following code:

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
  <jsp:directive.tag />
  <jsp:directive.attribute name="title" required="true" />
  <head>
    <meta http-equiv="content-type"
          content="text/html;charset=${pageCharSet}" />
    <title>${title}</title>
  </head>
</jsp:root>

where pageCharSet is a variable with a value as iso-8859-1.
```

Note that this tag is a JSP document (because of the jsp:root declaration), and, as such, it is validated by the container. Also note that the content that is generated in this case is not using QNames, which

means that the interpretation of the generated elements can be 'captured' based on the invocation context. That is the case here, as there is a default namespace active (that of XHTML) where the tag is being invoked.

6.5. Possible Future Directions for JSP documents

This section is non-normative. Two features are sketched briefly here to elicit input that could be used on future versions of the JSP specification.

6.5.1. Generating XML Content Natively

All JSP 3.0 content is textual, even when using JSP documents to generate XML content. This is quite acceptable, and even ideal, for some applications, but in some other applications XML documents are the main data type being manipulated. For example, the data source may be an XML document repository, perhaps queried using XQuery, some of the manipulation on this data internal to the JSP page will use XML concepts (XPath, XSTL operations), and the generated XML document may be part of some XML pipeline.

In one such application, it is appealing not to transform back and forth between a stream of characters (text) and a parsed representation of the XML document. The JSP expert group has explored different approaches on how such XML-awareness could be added, and a future version of JSP could support this functionality.

6.5.2. Schema and XInclude Support

The current specification only requires DTD validation support for JSP documents. A more flexible schema language, like XML Schema, could be useful and could be explored by a future version of the JSP specification.

Similarly, future versions of the specification may also consider support for XInclude.

Chapter 7. Tag Extensions

This chapter describes the tag library facility for introducing new actions into a JSP page. The tag library facility includes portable run-time support, a validation mechanism, and authoring tool support. Both the classic JSP 1.2 style tag extension mechanism and the newer JSP 2.0 onwards simple tag extension mechanism are described. In Chapter 8, *Tag Files*, a mechanism for authoring tag extensions using only JSP syntax is described. This brings the power of tag extensions to page authors that may not know the Java programming language.

This chapter also provides an overview of the tag library concept. It describes the Tag Library Descriptor, and the `taglib` directive. A detailed description of the APIs involved may be found in the `jakarta.servlet.jsp.tagext` Javadoc.

7.1. Introduction

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

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

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

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

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

The tag extension mechanism includes information to:

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

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

below.

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.

7.1.2. Overview

The processing of a JSP page conceptually follows these steps:

Parsing

JSP pages can be authored using two different syntaxes: a JSP syntax and an XML syntax. The semantics and validation of a JSP syntax page is described with reference to the semantics and validation of an equivalent document in the XML syntax.

The first step is to parse the JSP page. The page that is parsed is as expanded by the processing of include directives. Information in the TLD is used in this step, including the identification of custom tags, so there is some processing of the taglib directives in the JSP page.

Validation

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

Each library is checked for a validator class. If one is present, the whole document is made available to its `validate` method as a `PageData` object. As of JSP 2.0, the Container must provide a `jsp:id` attribute. This information can be used to provide location information on errors.

Each custom tag in the library is checked for a `TagExtraInfo` class. If one is present, its `validate` method is invoked. The default implementation of `validate` is to call `isValid`. See the APIs for more details.

Translation

Finally, the XML document is processed to create a JSP page implementation class. This process may

involve creating scripting variables. Each custom action will provide information about variables, either statically in the TLD, or more flexibly by using the `getVariableInfo` method of a `TagExtraInfo` class.

Execution

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

7.1.3. Classic Tag Handlers

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

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

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

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

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

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

7.1.4. Simple Examples of Classic Tag Handlers

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

7.1.4.1. Plain Actions

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

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

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

7.1.4.2. Actions with a Body

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

7.1.4.3. Conditionals

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

7.1.4.4. Iterations

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

7.1.4.5. Actions that Process their Body

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

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

7.1.4.6. Cooperating Actions

Cooperating actions may offer the best way to describe a desired functionality. For example, one action may be used to describe information leading to the creation of a server-side object, while another

action may use that object elsewhere in the page. These actions may cooperate explicitly, via scoped variables: one action creates an object and gives it a name; the other refers to the object through the name.

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

7.1.4.7. Actions Defining Scripting Variables

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

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

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

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

There are some sections of JSP where scripting is not allowed. For example, this is the case in a tag body where the `body-content` is declared as ‘`scriptless`’, or in a page where `<scripting-invalid>` is true. In these sections, it is not possible to access scripting variables directly via scriptlets or expressions, and therefore the container need not synchronize them. Instead, the page author can use the EL to access the `pageContext` values.

7.1.5. Simple Tag Handlers

The API and invocation protocol for classic tag handlers is necessarily somewhat complex because scriptlets and scriptlet expressions in tag bodies can rely on surrounding context defined using scriptlets in the enclosing page.

With the advent of the Expression Language (EL) and JSP Standard Tag Library (JSTL), it is now feasible to develop JSP pages that do not need scriptlets or scriptlet expressions. This allows us to define a tag invocation protocol that is easier to use for many use cases.

In that interest, JSP 2.0 introduced a new type of tag extension called a Simple Tag Extension. Simple Tag Extensions can be written in one of two ways:

- In Java, by defining a class that implements the `jakarta.servlet.jsp.tagext.SimpleTag` interface. This class is intended for use by advanced page authors and tag library developers who need the

flexibility of the Java language in order to write their tag handlers. The `jakarta.servlet.jsp.tagext.SimpleTagSupport` class provides a default implementation for all methods in `SimpleTag`.

- In JSP, using tag files. This method can be used by page authors who do not know Java. It can also be used by advanced page authors or tag library developers who know Java but are producing tag libraries that are presentation-centric or can take advantage of existing tag libraries. See Chapter 8, *Tag Files* for more details.

The lifecycle of a Simple Tag Handler is straightforward and is not complicated by caching semantics. Once a Simple Tag Handler is instantiated by the Container, it is executed and then discarded. The same instance must not be cached and reused. Initial performance metrics show that caching a tag handler instance does not necessarily lead to greater performance, and to accommodate such caching makes writing portable tag handlers difficult and makes the tag handler prone to error.

In addition to being simpler to work with, Simple Tag Extensions do not directly rely on any servlet APIs, which allows for potential future integration with other technologies. This is facilitated by the `JspContext` class, which `PageContext` now extends. `JspContext` provides generic services such as storing the `JspWriter` and keeping track of scoped attributes, whereas `PageContext` has functionality specific to serving JSPs in the context of servlets. Whereas the `Tag` interface relies on `PageContext`, `SimpleTag` only relies on `JspContext`.

The body of a Simple Tag, if present, is translated into a JSP Fragment and passed to the `setJspBody` method. The tag can then execute the fragment as many times as needed. See Section 7.1.6, "JSP Fragments" for more details on JSP Fragments.

Because JSP Fragments do not support scriptlets, the <body-content> of a SimpleTag cannot be "JSP". A JSP page is invalid if it references a custom tag whose tag handler implements the SimpleTag interface and whose <body-content> is equal to "JSP" as per the supporting TLD.

7.1.6. JSP Fragments

During the translation phase, various pieces of the page are translated into implementations of the `jakarta.servlet.jsp.tagext.JspFragment` abstract class, before being passed to a tag handler. This is done automatically for any JSP code in the body of a named attribute (one that is defined by `<jsp:attribute>`) that is declared to be a fragment, or of type `JspFragment`, in the TLD. This is also automatically done for the body of any tag handled by a Simple Tag handler. Once passed in, the tag handler can then evaluate and re-evaluate the fragment as many times as needed, or even pass it along to other tag handlers, in the case of Tag Files.

A JSP fragment can be parameterized by a tag handler by setting page-scoped attributes in the `JspContext` associated with the fragment. These attributes can then be accessed via the EL.

A translation error must occur if a piece of JSP code that is to be translated into a JSP Fragment contains scriptlets or scriptlet expressions.

See the `jakarta.servlet.jsp.tagext` Javadoc for more details on the `JspFragment` abstract class.

7.1.7. Simple Examples of Simple Tag Handlers

In this section, we revisit the prototypical uses of classic tag extensions, as was presented in Section 7.1.4, “Simple Examples of Classic Tag Handlers”, and briefly describe how they are implemented using simple tag handlers.

7.1.7.1. Plain Actions

To implement plain actions, the tag library developer creates a class that extends `SimpleTagSupport` and implements the `doTag` method. The details on accessing attributes and enforcing an empty body are the same as with classic tag handlers. By default, the rest of the page will be evaluated after invoking `doTag`. To signal that the page is to be skipped, `doTag` throws `SkipPageException`.

7.1.7.2. Actions with a Body

To implement actions with a body, the tag library developer implements `doTag` and invokes the body at any point by calling `invoke` on the `JspFragment` object passed in via the `setJspBody` method. The tag handler can provide the fragment access to variables through the `JspContext` object.

7.1.7.3. Conditionals

All conditional logic is handled in the `doTag` method. If the body is not to be invoked, the tag library developer simply does not call `invoke` on the `JspFragment` object passed in via `setJspBody`.

7.1.7.4. Iterations

All iteration logic is handled in the `doTag` method. The tag library developer simply calls `invoke` on the `JspFragment` object passed in via `setJspBody` as many times as needed.

7.1.7.5. Actions that Process their Body

To divert the result of the body invocation, the tag library developer passes a `java.io.Writer` object to the `invoke` method on the body `JspFragment`. Unlike the standard tag handler’s `BodyContent` solution, the result of the invocation does not need to be buffered.

7.1.7.6. Cooperating Actions

Cooperating actions work the same way as with classic tag handlers. A `setParent` method is available in the `SimpleTag` interface and is called by the container before calling `doTag` if one tag invocation is nested within another. A `findAncestorWithClass` method is available on `SimpleTagSupport`. This should be used, instead of `TagSupport.findAncestorWithClass(),` in all cases where the desired return value may implement `SimpleTag`.

Note that `SimpleTag` does not extend `Tag`. Because of this, the `JspTag` common base is used in these new APIs instead of `Tag`. Furthermore, because `Tag.setParent` only accepts an object of type `Tag`, tag collaboration becomes more difficult when classic tag handlers are nested inside `SimpleTag` custom actions.

To make things easier, the `jakarta.servlet.jsp.tagext.TagAdapter` class can wrap any `SimpleTag` and expose it as if it were a `Tag` instace. The original `JspTag` can be retrieved through its `getAdaptee` method. Whenever calling the `setParent` method on a classic `Tag` in a case where the outer tag does not implement `Tag`, the JSP Container must construct a new `TagAdapter` and call `setParent` on the classic `Tag` passing in the adapter.

See the `jakarta.servlet.jsp.tagext` Javadoc for more details on these APIs.

7.1.8. Attributes With Dynamic Names

Prior to JSP 2.0, the name of every attribute that a tag handler accepted was predetermined at the time the tag handler was developed. It is sometimes useful, however, to be able to define a tag handler that accepts attributes with dynamic names that are not known until the page author uses the tag. For example, it is time consuming and error-prone to anticipate what attributes a user may wish to pass to a tag that mimics an HTML element.

Available since JSP 2.0 is the ability to declare that a tag handler accepts additional attributes with dynamic names. This is done by having the tag handler implement the `jakarta.servlet.jsp.tagext.DynamicAttributes` interface. See the `jakarta.servlet.jsp.tagext` Javadoc for more details on this interface.

7.1.9. Event Listeners

A tag library may include classes that are event listeners (see the Servlet 5.0 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 7.3.1, “Identifying Tag Library Descriptors” 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.

7.1.10. JspId Attribute

Sometimes it may be useful to provide unique identifications for tag handlers. A tag handler can implement the interface `jakarta.servlet.jsp.tagext.JspIdConsumer` for such functionality. See `jakarta.servlet.jsp.tagext` Javadoc for more details.

7.1.11. Resource Injection

The Java Metadata specification (JSR-175), which is part of Java SE 5.0 and greater, provides a means of specifying configuration data in Java code. Metadata in Java code is also referred to as annotations. In Jakarta EE, annotations are used to declare dependencies on external resources and configuration data in Java code without the need to define that data in a configuration file.

Section SRV.15.5 of the Servlet Specification describes the behavior of annotations and resource

injection in Jakarta EE technology compliant web containers.

In the JSP specification, tag handlers which implement interfaces `jakarta.servlet.jsp.tagext.Tag` and `jakarta.servlet.jsp.tagext.SimpleTag` may be annotated for injection. In both cases, injection occurs immediately after an instance of the tag handler is constructed, and before any of the tag properties are initialized.

Event Listeners (See Section 7.1.9, “Event Listeners”) can also be annotated for resource injection. Injection occurs immediately after an instance of the event handler is constructed, and before it is registered.

The annotations supported are:

- `@EJB`, `@EJBs`
- `@PersistenceContext`, `@PersistenceContexts`
- `@PersistenceUnit`, `@PersistenceUnits`
- `@PostConstruct`, `@PreDestroy`
- `@Resource`, `@Resources`
- `@WebServiceRef`, `@WebServiceRefs`

Please see Section SRV.15.5 of the servlet specification for more details on these annotations.

A JSP container that is not part of a Jakarta EE technology-compliant implementation is encouraged, but not required, to support resource injection.

Resource injection is not supported for JSP pages or tag files.

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 7.3, “The Tag Library Descriptor” below.

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 5.0 specification apply, including the conventions for dependencies on extensions.

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

Both Classic and Simple Tag Handlers (implemented either in Java or as tag files) can be packaged together.

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 includes classes such as `TagLibraryValidator` and `TagExtraInfo`. The latter includes tag handler classes.

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

A JAR containing packaged tag libraries must be dropped into the `WEB-INF/lib` directory to make its classes available at request time (and also at translation time, see Section 7.3.7, "Translation-Time Class Loader"). The mapping between the URI and the TLD is explained further below.

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.

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 Jakarta EE technologies.

As of JSP 2.0, the format for the Tag Library Descriptor is represented in XML Schema. This allows for a more extensible TLD that can be used as a true single-source document.

7.3.1. Identifying Tag Library Descriptors

Tag library descriptor files have names that use the extension `.tld`, and the extension indicates a tag library descriptor file. When deployed inside a JAR file, the tag library descriptor files must be in the `META-INF` directory, or a subdirectory of it. When deployed directly into a web application, the tag library descriptor files must always be in the `WEB-INF` directory, or some subdirectory of it. TLD files should not be placed in `/WEB-INF/classes` or `/WEB-INF/lib`, and must not be placed inside `/WEB-INF/tags` or a subdirectory of it, unless named `implicit.tld` and intended to configure an implicit tag library with its JSP version and `tlib-version`.

The XML Schema for a TLD document is `http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_1.xsd`.

Note that tag files, which collectively form tag libraries, may or may not have an explicitly defined TLD. In the case that they do not, the container generates an implicit TLD that can be referenced using the `tagdir` attribute of the `taglib` directive. More details about identifying this implicit Tag Library Descriptor can be found in Chapter 8, *Tag Files*.

7.3.2. TLD Resource Path

A URI in a `taglib` directive is mapped into a context relative path (as discussed in Section 1.2.1, “Relative URL Specifications”). 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 7.3.3, “Taglib Map in web.xml”) that is extended

with implicit entries deduced from packaged tag libraries in the web application (described in Section 7.3.4, "Implicit Map Entries from TLDs"), 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 7.3.6.2, "Computing the TLD Resource Path").

The following order of precedence applies (from highest to lowest) when building the taglib map (see the following sections for details):

1. If the container is Jakarta EE platform compliant, the Map Entries for the tag libraries that are part of the Jakarta EE platform. This currently includes the Jakarta Server Pages Standard Tag Library libraries and the Jakarta Server Faces libraries.
2. Taglib Map in web.xml
3. Implicit Map Entries from TLDs
    a. TLDs in JAR files in WEB-INF/lib
    b. TLDs under WEB-INF
4. Implicit Map Entries from the Container

7.3.3. Taglib Map in web.xml

The `web.xml` file can include an explicit taglib map between URIs and TLD resource paths described using the `taglib` elements of the Web Application Deployment descriptor in `WEB-INF/web.xml`. See Section 3.2, "Taglib Map" for more details.

7.3.4. Implicit Map Entries from TLDs

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

- The container searches for all files with a `.tld` extension under `/WEB-INF` or a subdirectory, and inside JAR files that are in `/WEB-INF/lib`. When examining a JAR file, only resources under `/META-INF` or a subdirectory are considered. The order in which these files are searched for is implementation-specific and should not be relied on by web applications.
- Each TLD file is examined. If it has a `<uri>` element, then a new `<taglib>` element is created, with a `<taglib-uri>` subelement whose value is that of the `<uri>` element, and with a `<taglib-location>` subelement that refers to the TLD file.
- If the created `<taglib>` element has a different `<taglib-uri>` to any in the taglib map, it is added.

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

Note also that the mechanism does not add duplicated entries.

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 7.3.9, “Well-Known URIs”.

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.

7.3.6.1. Computing TLD Locations

The taglib map generated in Sections Section 7.3.3, “Taglib Map in web.xml”, Section 7.3.4, “Implicit Map Entries from TLDs” and Section 7.3.5, “Implicit Map Entries from the Container” 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 `/`).

7.3.6.2. Computing the TLD Resource Path

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

- If `uri` is `abs_uri`, an absolute URI

  Look in the taglib map for an entry whose `taglib_uri` is `abs_uri`. If found, the corresponding `taglib_location` is the TLD resource path. If not found, a translation error is raised.

- If `uri` is `root_rel_uri`, a relative URI that starts with `/`

  Look in the taglib map for an entry whose `taglib_uri` is `root_rel_uri`. If found, the corresponding `taglib_location` is the TLD resource path. If no such entry is found, `root_rel_uri` is the TLD

resource path.

- If `uri` is `noroot_rel_uri`, a relative URI that does not start with `/`

  Look in the `taglib map` for an entry whose `taglib_uri` is `noroot_rel_uri`. If found, the corresponding `taglib_location` is the TLD resource path. If no such entry is found, resolve `noroot_rel_uri` relative to the current JSP page where the directive appears; that value (by definition, this is a relative URI specification that starts with `/`) is the TLD resource path. For example, if `/a/b/c.jsp` references `../../WEB-INF/my.tld`, then if there is no `taglib_location` that matches `../../WEB-INF/my.tld`, the TLD resource path would be `/WEB-INF/my.tld`.

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" %>
```

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.

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 7.5.3, “Customizing a Tag Library”.

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.

7.3.10. Tag and Tag Library Extension Elements

The JSP 2.0 Tag Library Descriptor supports the notion of Tag Extension Elements and Tag Library Extension Elements. These are elements added to the TLD by the tag library developer that provide additional information about the tag, using a schema defined outside of the JSP specification.

The information contained in these extensions is intended to be used by tools only, and is not accessible at compile-time, deployment-time, or run-time. JSP containers must not alter their behavior based on the content, the presence, or the absence of a particular Tag or Tag Library Extension Element. In addition, JSP containers must consider invalid any tag library that specifies `mustUnderstand="true"` for any Tag or Tag Library Extension element. Any attempt to use an invalid tag library must produce a translation error. This is to preserve application compatibility across containers.

The JSP container may use schema to validate the structure of the Tag Library Descriptor. If it does so, any new content injected into Tag or Tag Library Extension elements must not be validated by the JSP Container.

Tag Library Extension Elements provide extension information at the tag library level, and are specified by adding a `<taglib-extension>` element as a child of `<taglib>`. Tag Extension Elements provide extension information at the tag level, and are specified by adding a `<tag-extension>` element

as a child of <tag>. To use these elements, an XML namespace must first be defined and the namespace must be imported into the TLD.

7.3.10.1. Example

In the following non-normative example, a fictitious company called ACME has decided to enhance the page author's experience by defining a set of Tag and Tag Library Extension elements that cause sounds to be played when inserting tags in a document.

In this hypothetical example, ACME has published an XML Schema at http://www.acme.com/acme.xsd that defines the extensions, and has provided plug-ins for various JSP-capable IDEs to recognize these extension elements.

The following example tag library uses ACME's extensions to provide helpful voice annotations that describe how to use each tag in the tag library.

```
<taglib xmlns="http://java.sun.com/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:acme="http://acme.com/"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
            http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_1.xsd
            http://acme.com/ http://acme.com/acme.xsd"
        version="2.1">
  <description>
    Simple Math Tag Library.
    Contains ACME sound extensions with helpful voice annotations
    that describe how to use the tags in this library.
  </description>
  <tlib-version>1.0</tlib-version>
  <short-name>math</short-name>
  <tag>
    <description>Adds two numbers</description>
    <display-name>add</display-name>
    <name>add</name>
    <tag-class>com.foobar.tags.AddTag</tag-class>
    <body-content>empty</body-content>
    <attribute>
      <name>x</name>
      <type>java.lang.Double</type>
    </attribute>
    <attribute>
    <name>y</name>
     <type>java.lang.Double</type>
    </attribute>
    <tag-extension namespace="http://acme.com/">
      <!-- Extensions for tag sounds -->
      <extension-element xsi:type="acme:acme-soundsType">
```

```
        <acme:version>1.5</acme:version>
        <!-- Sound played for help on the add tag -->
        <acme:tag-sound>sounds/add.au</acme:tag-sound>
        <!-- Sound played for help on the x attribute -->
        <acme:attribute-sound name="x">
          sounds/add-x.au
        </acme:attribute-sound>
        <!-- Sound that's played for help on the yattribute -->
        <acme:attribute-sound name="y">
          sounds/add-y.au
        </acme:attribute-sound>
      </extension-element>
    </tag-extension>
  </tag>
  <taglib-extension namespace="http://acme.com/">
    <!-- Extensions for taglibrary sounds-->
    <extension-element xsi:type="acme:acme-soundsType">
      <acme:version>1.5</acme:version>
      <!-- Sound played when author imports this taglib -->
      <acme:import-sound>sounds/intro.au</acme:import-sound>
    </extension-element>
  </taglib-extension>
</taglib>
```

The corresponding `acme.xsd` file would look something like:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="http://acme.com/"
            xmlns:j2ee="http://java.sun.com/xml/ns/j2ee"
            xmlns:acme="http://acme.com/"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:xml="http://www.w3.org/XML/1998/namespace"
            elementFormDefault="qualified"
            attributeFormDefault="unqualified"
            version="1.0">
  <xsd:annotation>
    <xsd:documentation>
      This an XML Schema for sample Acme taglib extensibility
       elements, used to test TLD extensibility.
    </xsd:documentation>
  </xsd:annotation>

  <!-- ************************************************** -->

  <xsd:import namespace="http://java.sun.com/xml/ns/j2ee"
              schemaLocation="../web-jsptaglibrary_2_0.xsd" />
```

```
  <!-- ************************************************* -->

  <xsd:complexType name="acme-soundsType">
    <xsd:annotation>
      <xsd:documentation>
        Extension for sounds associated with a tag
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="j2ee:extensibleType">
        <xsd:sequence>
          <xsd:element name="version" type="xsd:string"/>
          <xsd:element name="tag-sound" type="xsd:string"
                       minOccurs="0" maxOccurs="unbounded"/>
          <xsd:element name="attribute-sound"
                       minOccurs="0" maxOccurs="unbounded">
            <xsd:complexType>
              <xsd:simpleContent>
                <xsd:extension base="xsd:string">
                  <xsd:attribute name="name" use="required" type="xsd:string" />
                </xsd:extension>
              </xsd:simpleContent>
            </xsd:complexType>
          </xsd:element>
          <xsd:element name="import-sound" type="xsd:string"
                       minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:schema>
```

7.4. Validation

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

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

Validation can be done either at translation-time or at request-time. In general translation-time validation provides a better user experience, and the JSP 3.0 specification provides a very flexible

translation-time validation mechanism.

7.4.1. Translation-Time Mechanisms

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

7.4.1.1. Attribute Information

The Tag Library Descriptor contains the basic syntactic information. In particular, the attributes are described including their name, whether they are optional or mandatory, and whether they accept request-time expressions. Additionally the `body-content` element can be used to indicate that an action must be empty.

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

7.4.1.2. Validator Classes

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

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

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.

7.4.1.3. TagExtraInfo Class Validation

Additional translation-time validation can be done using the `validate` method in the `TagExtraInfo` class. The `validate` method is invoked at translation-time and is passed a `TagData` instance as its argument. As of JSP 2.0, the default behavior of `validate` is to call the `isValid` method.

The `isValid` mechanism was the original validation mechanism introduced in JSP 1.1 with the rest of the Tag Extension machinery. Tag libraries that are designed to run in JSP 1.2 containers or higher should use the validator class mechanism. Tag libraries that are designed to run in JSP 2.0 containers or higher that wish to use the `TagExtraInfo` validation mechanism are encouraged to implement the `validate` method in favor of the `isValid` method due to the ability to provide better validation messages. Either method will work, though implementing both is not recommended.

7.4.2. Request-Time Errors

In some cases, additional request-time validation will be done dynamically within the methods in the tag handler. If an error is discovered, an instance of `JspException` can be thrown. If uncaught, this object will invoke the errorpage mechanism of the JSP specification.

7.5. Conventions and Other Issues

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

7.5.1. How to Define New Implicit Objects

We advocate the following style for the introduction of implicit objects:

- Define a tag library.
- Add an action called `defineObjects` to define the desired objects.

The JSP page can make these objects available as follows:

```
<%@ taglib prefix="me" uri="..." %>
<me:defineObjects />
... start using the objects...
```

This approach has the advantage of requiring no new machinery and of making very explicit the dependency.

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

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

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.

7.5.2. Access to Vendor-Specific information

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

document in the `WEB-INF` portion of the Web Application where the Tag Library resides, and using the standard Servlet 5.0 mechanisms to access that information.

7.5.3. Customizing a Tag Library

A tag library can be customized at assembly and deployment time. For example, a tag library that provides access to databases may be customized with login and password information.

There is no convenient place in `web.xml` in the Servlet 5.0 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 8. Tag Files

This chapter describes the details of tag files, a JSP 2.0 onwards facility that allows page authors to author tag extensions using only JSP syntax. In the past, the ability to encapsulate presentation logic into reusable, full-featured tag libraries was only available to developers that had a reasonable amount of Java experience. Tag files bring the power of reuse to the basic page author, who are not required to know Java. When used together with JSP Fragments and Simple Tag Handlers, these concepts have the ability to simplify JSP development substantially, even for developers who do know Java.

8.1. Overview

As of JSP version 2.0, the JSP Compiler is required to recognize tag files. A tag file is a source file that provides a way for a page author to abstract a segment of JSP code and make it reusable via a custom action.

Tag files allow a JSP page author to create tag libraries using JSP syntax. This means that page authors no longer need to know Java or ask someone who knows Java to write a tag extension. Even for page authors or tag library developers who know Java, writing tag files is more convenient when developing tags that primarily output template text.

The required file extension for a tag file are `.tag` or `.tagx`. As is the case with JSP files, the actual tag may be composed of a top file that includes other files that contain either a complete tag or a segment of a tag file. Just as the recommended extension for a segment of a JSP file is `.jspf`, the recommended extension for a segment of a tag file is `.tagf`.

8.2. Syntax of Tag Files

The syntax of a tag file is similar to that of a JSP page, with the following exceptions:

- Directives - Some directives are not available or have limited availability, and some tag file specific directives are available. See Section 8.5, “Tag File Directives” for a discussion on tag file directives.
- The `<jsp:invoke>` and `<jsp:doBody>` standard actions can only be used in Tag Files.

The EBNF grammar in Section 1.3.10, “JSP Syntax Grammar” describes the syntax of tag files. The root production for a tag files is `JSPTagDef`.

See Section 8.6, “Tag Files in XML Syntax” for details on tag files in XML syntax.

8.3. Semantics of Tag Files

For each tag file in the web application, a tag handler is made available to JSP pages and other tag files. The specifics of how this is done are left up to the Container implementation. For example, some Containers may choose to compile tag files into Java tag handlers, whereas others may decide to interpret the tag handlers.

However the Container chooses to prepare the tag handler, the following conditions must hold true for all tag handlers defined as tag files:

- The tag file implementation must keep a copy of the `JspContext` instance passed to it by the invoking page via the `setJspContext` method. This is called the *Invoking JSP Context*.
- The tag file implementation must create and maintain a second instance of `JspContext` called a *JSP Context Wrapper*. If the Invoking JSP Context is an instance of `PageContext`, the JSP Context Wrapper must also be an instance of `PageContext`. This wrapper must be returned when `getJspContext()` is called.
- For each invocation to the tag, the JSP Context Wrapper must present a clean page scope containing no initial elements. All scopes other than the page scope must be identical to those in the Invoking JSP Context and must be modified accordingly when updates are made to those scopes in the JSP Context Wrapper. Any modifications to the page scope, however, must not affect the Invoking JSP Context.
- For each attribute declared and specified, a page-scoped variable must be created in the page scope of the JSP Context Wrapper, unless the attribute is a deferred value or a deferred method, in which case the VariableMapper obtained from the ELContext in the current pageContext is used to map the deferred expression to the attribute name. The name of the variable must be the same as the declared attribute name. The value of the variable must be the value of the attribute passed in during invocation. For each attribute declared as optional and not specified, no variable is created. If the tag accepts dynamic attributes, then the names and values of those dynamic attributes must be exposed to the tag file as specified in **Table JSP.8-2** , “Details of `tag` directive attributes”.

  If the attribute is a deferred-value, it is directly mapped. If the attribute is a deferred-method, it is wrapped in a `ValueExpression`, and the resulting `ValueExpression` is mapped.

  There are two implications here. They are best illustrated by examples. Suppose we have a tag file tagf.tag:

```
<%@ attribute name="attr1" deferredValue="true"/>
<%@ attribute name="attr2" deferredMethod="true"/>
<c:out value="${attr1.bar}"/>
<h:commandButton value="#{attr1.foo}" action="#{attr2}"/>
```

  used in test.jsp:

```
<%@ taglib prefix="my" tagdir="/WEB-INF/tags"%>
<my:tagf attr1="#{someExpr}" attr2="#{someMethod}"/>
```

  First, in `tagf.tag`, `${attr1.bar}` will cause the immediate evaluation of the deferred expression. Secondly, since the `VariableMapper` is used to resolve variables at EL parse time, a deferred expression such as `#{attr1.foo}` is not dependent on `attr1` anymore, so that it can be evaluated long after the end of life of the tag file’s pageContext. This is very useful for Jakarta Server Faces

applications.

Since the EL syntax does not allow for invocation of the method in a `MethodExpression`, the only allowable use of `attr2` is to pass it to another tag that has a deferred-method attribute, in the form of "#{attr2}".

- For all intents and purposes other than for synchronizing the `AT_BEGIN`, `NESTED`, and `AT_END` scripting variables, the effective `JspContext` for the tag file is the JSP Context Wrapper. For example, the `jspContext` scripting variable must point to the JSP Context Wrapper instead of the invoking JSP Context.
- The tag handler must behave as though a tag library descriptor entry was defined for it, in accordance with the `tag`, `attribute`, and `variable` directives that appear in the tag file translation unit.

It is legal for a tag file to forward to a page via the `<jsp:forward>` standard action. Just as for JSP pages, the forward is handled through the request dispatcher. Upon return from the `RequestDispatcher.forward` method, the generated tag handler must stop processing of the tag file and throw `jakarta.servlet.jsp.SkipPageException`. Similarly, if a tag file invokes a Classic Tag Handler which returns `SKIP_PAGE` from the `doEndTag` method, or if it invokes a Simple Tag Handler which throws `SkipPageException` in the `doTag` method, the generated tag handler must terminate and `SkipPageException` must be thrown. In either of these two cases, the `doCatch` and `doFinally` methods must be called on enclosing tags that implement the `TryCatchFinally` interface before returning. The `doEndTag` methods of enclosing classic tags must not be called.

Care should be taken when invoking a classic tag handler from a tag file. In general, SimpleTag Extensions can be used in environments other than servlet environments. However, because the `Tag` interface relies on `PageContext`, which in turn assumes a servlet environment, using classic tag handlers indirectly binds the use of the tag file to servlet environments. Nonetheless, the JSP container must allow such an invocation to occur. When a tag file attempts to invoke a classic tag handler (i.e. one that implements the `Tag` interface), it must cast the `JspContext` passed to the `SimpleTag` into a `PageContext`. In the event that the class cast fails, the invocation of the classic tag fails, and a `JspException` must be thrown.

If a tag file in XML syntax contains a jsp:root element, the value of that element's version attribute must match the tag file's JSP version. See Section 8.4.2, "Packaging in a JAR", and Section 8.4.3, "Packaging Directly in a Web Application", for how the JSP version of a tag file is determined.

8.4. Packaging Tag Files

One of the goals of tag files as a technology is to make it as easy to write a tag handler as it is to write a JSP. Traditionally, writing tag handlers has been a tedious task, with a lot of time spent compiling and packaging the tag handlers and writing a TLD to provide information to tools and page authors about the custom actions. The rules for packaging tag files are designed to make it very simple and fast to write simple tags, while still providing as much power and flexibility as classic tag handlers have.

8.4.1. Location of Tag Files

Tag extensions written in JSP using tag files can be placed in one of two locations. The first possibility is in the `/META-INF/tags/` directory (or a subdirectory of `/META-INF/tags/`) in a `JAR` file installed in the `/WEB-INF/lib/` directory of the web application. Tags placed here are typically part of a reusable library of tags that can be easily dropped into any web application.

The second possibility is in the `/WEB-INF/tags/` directory (or a subdirectory of `/WEB-INF/tags/`) of the web application. Tags placed here are within easy reach and require little packaging. Only files with a `.tag` or `.tagx` extension are recognized by the container to be tag files.

Tag files that appear in any other location are not considered tag extensions and must be ignored by the JSP container. For example, a tag file that appears in the root of a web application would be treated as content to be served.

8.4.2. Packaging in a JAR

To be accessible, tag files bundled in a `JAR` require a Tag Library Descriptor. Tag files that appear in a JAR but are not defined in a TLD must be ignored by the JSP container.

JSP 2.0 added an additional TLD element to describe tags within a tag library, namely `<tag-file>`. The `<tag-file>` element requires `<name>` and `<path>` subelements, which define the tag name and the full path of the tag file from the root of the JAR, respectively. In a JAR file, the `<path>` element must always begin with `/META-INF/tags`. The values for the other subelements of `<tag-file>` override the defaults specified in the tag directive. Tag files packaged in a JAR inherit the JSP version of the TLD that references them.

Note that it is possible to combine both classic tag handlers and tag handlers implemented using tag files in the same tag library by combining the use of `<tag>` and `<tag-file>` elements under the `<taglib>` element. This means that in most instances the client is unaware of how the tag extension was implemented. Given that `<tag>` and `<tag-file>` share a namespace, a tag library is considered invalid and must be rejected by the container if a `<tag-file>` element has a `<name>` subelement with the same content as a `<name>` subelement in a `<tag>` element. Any attempt to use an invalid tag library must trigger a translation error.

8.4.3. Packaging Directly in a Web Application

Tag files placed in the `/WEB-INF/tags/` directory of the web application, or a subdirectory, are made easily accessible to JSPs without the need to explicitly write a Tag Library Descriptor. This makes it convenient for page authors to quickly abstract reusable JSP code by simply creating a new file and placing the code inside of it.

The JSP container must interpret the `/WEB-INF/tags/` directory and each subdirectory under it, as another implicitly defined tag library containing tag handlers defined by the tag files that appear in that directory. There are no special relationships between subdirectories - they are allowed simply for organizational purposes. For example, the following web application contains three tag libraries:

```
/WEB-INF/tags/
/WEB-INF/tags/a.tag
/WEB-INF/tags/b.tag
/WEB-INF/tags/foo/
/WEB-INF/tags/foo/c.tagx
/WEB-INF/tags/bar/baz/
/WEB-INF/tags/bar/baz/d.tag
```

The JSP container must generate an implicit tag library for each directory under and including `/WEB-INF/tags/`. This tag library can be imported only via the `tagdir` attribute of the `taglib` directive (see Section 1.10.2, "The `taglib` Directive"), and has the following hard-wired values:

- `<tlib-version>` for the tag library defaults to 1.0
- `<short-name>` is derived from the directory name. If the directory is `/WEB-INF/tags/`, the short name is simply `tags`. Otherwise, the full directory path (relative to the web application) is taken, minus the `/WEB-INF/tags/` prefix. Then, all `/` characters are replaced with `-`, which yields the short name. Note that short names are not guaranteed to be unique (as in `/WEB-INF/tags/` versus `/WEB-INF/tags/tags/` or `/WEB-INF/tags/a-b/` versus `/WEB-INF/tags/a/b/`)
- A `<tag-file>` element is considered to exist for each tag file in this directory, with the following sub-elements:
  - The `<name>` for each is the filename of the tag file, without the `.tag` or `.tagx` extension.
  - The `<path>` for each is the path of the tag file, relative to the root of the web application.

For the above example, the implicit Tag Library Descriptor for the `/WEB-INF/tags/bar/baz/` directory would be:

```
<taglib>
  <tlib-version>1.0</tlib-version>
  <short-name>bar-baz</short-name>
  <tag-file>
    <name>d</name>
    <path>/WEB-INF/tags/bar/baz/d.tag</path>
  </tag-file>
</taglib>
```

The JSP version of an implicit tag library defaults to 2.0.

The JSP version and tlib-version of an implicit tag library may be configured by placing a TLD with the reserved name implicit.tld in the same directory as the implicit tag library's constituent tag files. A JSP 2.1 onwards container must consider only the JSP version and tlib-version specified by an implicit.tld file, and ignore its short-name element. Any additional elements in an implicit.tld file must cause a translation error. The JSP version specified in an implicit.tld file must be equal to or greater than 2.0, or else a translation error must be reported.

Upon deployment, the JSP container must search for and process all tag files appearing in these directories and subdirectories. In processing a tag file, the container makes the custom actions defined in these tags available to JSP files.

If a directory contains two files with the same tag name (e.g. `a.tag` and `a.tagx`), it is considered to be the same as having a TLD file with two `<tag>` elements whose `<name>` sub-elements are identical. The tag library is therefore considered invalid.

Despite the existence of an implicit tag library, a Tag Library Descriptor in the web application can still create additional tags from the same tag files. This is accomplished by adding a `<tag-file>` element with a `<path>` that points to the tag file. In this case, the value of `<path>` must start with `/WEB-INF/tags`. It a tag file is referenced by both a TLD as well as an implicit TLD, the JSP versions of the TLD and implicit TLD do not need to match.

8.4.4. Packaging as Precompiled Tag Handlers

Tag files can also be compiled into Java classes and bundled as a tag library. This is useful for the situation where a tag library developer wishes to distribute a binary version of the tag library without the original source. Tag library developers that choose this form of packaging must use a tool that produces portable JSP code that uses only standard APIs. Containers are not required to provide such a tool.

8.5. Tag File Directives

This section describes the directives available within tag files, which define Simple Tag Handlers. **Table JSP.8-1** , “Directives available to tag files” outlines which directives are available in tag files:

Table JSP.8-1 *Directives available to tag files*

Directive	Available?	Interpretation/Restrictions
`page`	no	A tag file is not a `page`. The `tag` directive must be used instead. If this directive is used in a tag file, a translation error must result.
`taglib`	yes	Identical to JSP pages.
`include`	yes	Identical to JSP pages. Note that if the included file contains syntax unsuitable for tag files, a translation error must occur.
`tag`	yes	Only applicable to tag files. An attempt to use this directive in JSP pages will result in a translation error.
`attribute`	yes	Only applicable to tag files. An attempt to use this directive in JSP pages will result in a translation error.
`variable`	yes	Only applicable to tag files. An attempt to use this directive in JSP pages will result in a translation error.

8.5.1. The tag Directive

The `tag` directive is similar to the `page` directive, but applies to tag files instead of JSPs. Like the `page` directive, a translation unit can contain more than one instance of the `tag` directive, all the attributes will apply to the complete translation unit (i.e. `tag` directives are position independent). There shall be only one occurrence of any attribute/value defined by this directive in a given translation unit, unless the values for the duplicate attributes are identical for all occurrences. The `import` and `pageEncoding` attributes are exempt from this rule and can appear multiple times. Multiple uses of the `import` attribute are cumulative (with ordered set union semantics). Other such multiple attribute/value (re)definitions result in a fatal translation error if the values do not match.

The attribute/value namespace is reserved for use by this, and subsequent, JSP specifications.

Unrecognized attributes or values result in fatal translation errors.

Examples

```
<%@ tag display-name="Addition"
        body-content="scriptless"
        dynamic-attributes="dyn"
        small-icon="/WEB-INF/sample-small.jpg"
        large-icon="/WEB-INF/sample-large.jpg"
        description="Sample usage of tag directive" %>
```

Syntax

```
<%@ tag tag_directive_attr_list %>

tag_directive_attr_list ::= { display-name="display-name"                  }
                            { body-content="scriptless|tagdependent|empty" }
                            { dynamic-attributes="name"                    }
                            { small-icon="small-icon"                      }
                            { large-icon="large-icon"                      }
                            { description="description"                    }
                            { example="example"                            }
                            { language="scriptingLanguage"                 }
                            { import="importList"                          }
                            { pageEncoding="peinfo"                        }
                            { isELIgnored="true|false"                     }
                            { deferredSyntaxAllowedAsLiteral="true|false"  }
                            { trimDirectiveWhitespaces="true|false"        }
```

The details of the attributes are as follows:

Table JSP.8-2 *Details of* `tag` *directive attributes*

display-name	(optional) A short name that is intended to be displayed by tools. Defaults to the name of the tag file, without the .tag extension.
body-content	(optional) Provides information on the content of the body of this tag. Can be either empty, tagdependent , or scriptless. A translation error will result if JSP or any other value is used. Defaults to scriptless.
dynamic-attributes	(optional) The presence of this attribute indicates this tag supports additional attributes with dynamic names. If present, the generated tag handler must implement the jakarta.servlet.jsp.tagext.DynamicAttributes interface, and the container must treat the tag as if its corresponding TLD entry contained <dynamic-attributes>true</dynamic-attributes>. The implementation must not reject any attribute names. The value identifies a page scoped attribute in which to place a Map containing the names and values of the dynamic attributes passed during this invocation. The Map must contain each dynamic attribute name as the key and the dynamic attribute value as the corresponding value. Only dynamic attributes with no uri are to be present in the Map ; all other dynamic attributes are ignored. A translation error will result if there is a tag directive with a dynamic-attributes attribute equal to the value of a name-given attribute of a variable directive or equal to the value of a name attribute of an attribute directive in this translation unit.
small-icon	(optional) Either a context-relative path, or a path relative to the tag source file, of an image file containing a small icon that can be used by tools. Defaults to no small icon.
large-icon	(optional) Either a context-relative path, or a path relative to the tag source file, of an image file containing a large icon that can be used by tools. Defaults to no large icon.
description	(optional) Defines an arbitrary string that describes this tag. Defaults to no description.
example	(optional) Defines an arbitrary string that presents an informal description of an example of a use of this action. Defaults to no example.
language	(optional) Carries the same syntax and semantics of the language attribute of the page directive.
import	(optional) Carries the same syntax and semantics of the import attribute of the page directive.
pageEncoding	(optional) Carries the same syntax and semantics of the pageEncoding attribute in the page directive. However, there is no corresponding global configuration element in web.xml. The pageEncoding attribute cannot be used in tag files in XML syntax.
isELIgnored	(optional) Carries the same syntax and semantics of the isELIgnored attribute of the page directive. However, there is no corresponding global configuration element in web.xml.

deferredSyntax-AllowedAsLiteral	(optional) Carries the same syntax and semantics of the deferredSyntaxAllowedAsLiteral attribute of the page directive. However, there is no corresponding global configuration element in web.xml. Causes a translation error if specified in a tag file with a JSP version less than 2.1.
trimDirective-Whitespaces	(optional) Carries the same syntax and semantics of the trimDirectiveWhitespaces attribute of the page directive. However, there is no corresponding global configuration element in web.xml.

8.5.2. The attribute Directive

The attribute directive is analogous to the <attribute> element in the Tag Library Descriptor, and allows for the declaration of custom action attributes.

Examples

```
<%@ attribute name="x" required="true" fragment="false"
              rtexprvalue="false" type="java.lang.Integer"
              description="The first operand" %>
```

```
<%@ attribute name="y" type="java.lang.Integer" %>
```

```
<%@ attribute name="prompt" fragment="true" %>
```

Syntax

```
<%@ attribute attribute_directive_attr_list %>

attribute_directive_attr_list ::= name="attribute-name"
                                  { required="true|false"              }
                                  { fragment="true|false"              }
                                  { rtexprvalue="true|false"           }
                                  { type="type"                        }
                                  { description="description"          }
                                  { deferredValue="true|false"         }
                                  { deferredValueType="type"           }
                                  { deferredMethod="true|false"        }
                                  { deferredMethodSignature="signature" }
```

The details of the attributes are as follows:

Table JSP.8-3 *Details of* attribute *directive attributes*

`name`	(required) The unique name of the attribute being declared. A translation error must result if more than one `attribute` directive appears in the same translation unit with the same `name`. A translation error will result if there is an `attribute` directive with a `name` attribute equal to the value of the `name-given` attribute of a `variable` directive or the `dynamic-attributes` attribute of a `tag` directive in this translation unit.
`required`	(optional) Whether this attribute is required (`true`) or optional (`false`). Defaults to `false` if not specified.
`fragment`	(optional) Whether this attribute is a fragment to be evaluated by the tag handler (`true`) or a normal attribute to be evaluated by the container prior to being passed to the tag handler. If this attribute is `true`, the type attribute is fixed at `jakarta.servlet.jsp.tagext.JspFragment` and a translation error will result if the `type` attribute is specified. Also, if this attribute is `true`, the `rtexprvalue` attribute is fixed at `true` and a translation error will result if the `rtexprvalue` attribute is specified. Defaults to `false`.
`rtexprvalue`	(optional) Whether the attribute's value may be dynamically calculated at runtime by a scriptlet expression. Unlike the corresponding TLD element, this attribute defaults to `true`.
`type`	(optional) The runtime type of the attribute's value. Defaults to `java.lang.String` if not specified. It is a translation error to specify a primitive type.
`description`	(optional) Description of the attribute. Defaults to no description.
`deferredValue`	(optional) Whether the attribute's value represents a deferred value expression. Only one of `deferredValue` or `deferredMethod` may be true. If `deferredValueType` is specified, default is true, otherwise default is false. Causes a translation error if specified in a tag file with a JSP version less than 2.1.
`deferredValueType`	(optional) The expected type resulting from the evaluation of the attribute's value expression. If both `deferredValueType` and `deferredValue` are specified, `deferredValue` must be true. If `deferredValue` is true, default is `java.lang.Object` . Causes a translation error if specified in a tag file with a JSP version less than 2.1.
`deferredMethod`	(optional) Whether the attribute's value represents a deferred method expression. Only one of `deferredValue` or `deferredMethod` may be true. If `deferredMethodSignature` is specified, default is true, otherwise default is false. Causes a translation error if specified in a tag file with a JSP version less than 2.1.
`deferredMethod-Signature`	(optional) The signature, as defined in the Java Language Specification, of the method to be invoked in the attribute's method expression. If both `deferredMethod` and `deferredMethodSignature` are specified, `deferredMethod` must be true. If `deferredMethod` is true and `deferredMethodSignature` is not specified, it defaults to `void methodname()`. Causes a translation error if specified in a tag file with a JSP version less than 2.1.

8.5.3. The variable Directive

The `variable` directive is analogous to the `<variable>` element in the Tag Library descriptor, and defines the details of a variable exposed by the tag handler to the calling page.

See Section 7.1.4.7, "Actions Defining Scripting Variables" for more details.

Examples

```
<%@ variable name-given="sum"
             variable-class="java.lang.Integer"
             scope="NESTED"
             declare="true"
             description="The sum of the two operands" %>
```

```
<%@ variable name-given="op1"
             variable-class="java.lang.Integer"
             description="The first operand" %>
```

```
<%@ variable name-from-attribute="var" alias="result" %>
```

Syntax

```
<%@ variable variable_directive_attr_list %>

variable_directive_attr_list ::= (   name-given="output-name"
                                   | ( name-from-attribute="attr-name"
                                       alias="local-name"
                                     )
                                 )
                                 { variable-class="output-type"   }
                                 { declare="true|false"           }
                                 { scope="AT_BEGIN|AT_END|NESTED" }
                                 { description="description"      }
```

The details of the attributes are as follows:

Table JSP.8-4 *Details of* `variable` *directive attributes*

`name-given`	Defines a scripting variable to be defined in the page invoking this tag. Either the `name-given` attribute or the `name-from-attribute` attribute must be specified. Specifying neither or both will result in a translation error. A translation error will result if two `variable` directives have the same `name-given`. A translation error will result if there is a `variable` directive with a `name-given` attribute equal to the value of the `name` attribute of an `attribute` directive or the `dynamic-attributes` attribute of a `tag` directive in this translation unit.
`name-from-attribute`	Defines a scripting variable to be defined in the page invoking this tag. The specified name is the `name` of an attribute whose (translation-time) value at of the start of the tag invocation will give the name of the variable. A translation error will result if there is no `attribute` directive with a `name` attribute equal to the value of this attribute that is of type `java.lang.String` , is `required` and not an `rtexprvalue`. Either the `name-given` attribute or the `name-from-attribute` attribute must be specified. Specifying neither or both will result in a translation error. A translation error will result if two variable directives have the same `name-from-attribute`.
`alias`	Defines a locally scoped attribute to hold the value of this variable. The container will synchronize this value with the variable whose name is given in `name-from-attribute`. Required when `name-from-attribute` is specified. A translation error must occur if used without `name-from-attribute`. A translation error must occur if the value of `alias` is the same as the value of a `name` attribute of an `attribute` directive or the `alias or name-given` attribute of a `variable` directive in the same translation unit.
`variable-class`	(optional) The name of the class of the variable. The default is `java.lang.String`.
`declare`	(optional) Whether the variable is declared or not in the calling page/tag file, after this tag invocation. `true` is the default.
`scope`	(optional) The scope of the scripting variable defined. Can be either `AT_BEGIN`, `AT_END`, or `NESTED`. Defaults to `NESTED`.
`description`	(optional) An optional description of this variable. Defaults to no description.

8.6. Tag Files in XML Syntax

Tag files can be authored using the XML syntax, as described in Chapter 6, *JSP Documents*. This section describes the few distinctions from the case of JSP documents.

Tag files in XML syntax must have the extension `.tagx`. All files with extension `.tagx` according to the rules in Section 8.4.1, “Location of Tag Files” are tag files in XML syntax. Conversely, files with extension `.tag` are not in XML syntax.

The `jsp:root` element can, but needs not, appear in tag files in XML syntax. A `jsp:root` element cannot appear in a tag file in JSP syntax.

As indicated in Section 5.16, "<jsp:output>", the default for tag files, in either syntax, is not to generate the xml declaration. The element `jsp:output` can be used to change that default for tag files in XML syntax.

Finally, the `tag` directive in a tag file in XML syntax cannot include a `pageEncoding` attribute; the encoding is inferred using the conventions for XML documents. Using the `pageEncoding` attribute shall result in a translation-time error.

8.7. XML View of a Tag File

Similar to JSP pages, tag files have an equivalent XML document, the XML view of a tag file, that is exposed to the translation phase for validation. During the translation phase for a tag file, a tag XML view is created and passed to all tag library validators declared in all tag libraries declared in the tag file.

The XML view of a tag file is identical to the XML view of a JSP, except that there are additional XML elements defined to handle tag file specific features. The XML view of a tag file is obtained in the same way that the XML view of a JSP page is obtained (see Chapter 10, *XML View*).

8.8. Implicit Objects

Tag library developers writing tag files have access to certain implicit objects that are always available for use within scriptlets and expressions through scripting variables that are declared implicitly at the beginning of the tag handler implementation. All scripting languages are required to provide access to these objects.

Each implicit object has a class or interface type defined in a core Java technology or Jakarta Servlet API package, as shown in **Table JSP.8-5** , "Implicit Objects Available in Tag Files".

Table JSP.8-5 *Implicit Objects Available in Tag Files*

Variable Name	Type	Semantics & Scope

`request`	protocol dependent subtype of:
	`jakarta.servlet.ServletRequest`
	e.g:
	`jakarta.servlet.http.HttpServletRequest`
		The request triggering the service invocation.
		`request` scope.

`response`	protocol dependent subtype of:
	`jakarta.servlet.ServletResponse`
	e.g:
	`jakarta.servlet.http.HttpServletResponse`
		The response to the request.
		`page` scope.

Variable Name	Type	Semantics & Scope
`jspContext`	`jakarta.servlet.jsp.JspContext`	The `JspContext` for this tag file. `page` scope.
`session`	`jakarta.servlet.http.HttpSession`	The session object created for the requesting client (if any). This variable is only valid for HTTP protocols. `session` scope.
`application`	`jakarta.servlet.ServletContext`	The servlet context obtained from the servlet configuration object (as in the call `getServletConfig().getContext()`). `application` scope.
`out`	`jakarta.servlet.jsp.JspWriter`	An object that writes into the output stream. `page` scope.
`config`	`jakarta.servlet.ServletConfig`	The `ServletConfig` for this JSP page. `page` scope.

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

8.9. Variable Synchronization

Just as is the case for all tag handlers, a tag file is able to communicate with its calling page via variables. As mentioned earlier, in tag files, variables are declared using the `variable` directive. Though the scopes of variables are similar to those in classic tag handlers, the semantics are slightly different. The intent is to be able to emulate IN and OUT parameters using attributes and variables, which appear as page-scoped attributes local to the tag file, and are synchronized with the calling page's `JspContext` at various points.

The `name-from-attribute` and `alias` attributes of the `variable` directive can be used to allow the caller to customize the name of the variable in the calling page while referring to a constant name in the tag file. When using these attributes, the name of the variable in the calling page is derived from the value of `name-from-attribute` at the time the tag was called. The name of the corresponding variable in the tag file is the value of `alias`.

- IN parameters - Use attributes. For each attribute, a page-scoped attribute is made available in the `JspContext` of the tag file. The page-scoped attribute is initialized to the value of the attribute when the tag is called. No further synchronization is performed.
- OUT parameters - Use variables with scope `AT_BEGIN` or `AT_END`. For each `AT_BEGIN` or `AT_END` variable, a page-scoped attribute is made available in the `JspContext` of the tag file. The scoped attribute is

not initialized. Synchronization is performed at the end of the tag for `AT_BEGIN` and `AT_END` and also before the invocation of a fragment for `AT_BEGIN`. See **Table JSP.8-6** , "Variable synchronization behavior" for details.

- Nested parameters - Use variables with scope `AT_BEGIN` or `NESTED`. For each `AT_BEGIN` or `NESTED` variable, a page-scoped attribute is made available in the `JspContext` of the tag file. The scoped attribute is not initialized. Synchronization is performed before each fragment invocation for `AT_BEGIN` and `NESTED`, and also after the end of the tag for `AT_BEGIN`. See **Table JSP.8-6** , "Variable synchronization behavior" for details.

8.9.1. Synchronization Points

The JSP container is required to generate code to handle the synchronization of each declared variable. The details of how and when each variable is synchronized varies by the variable's scope, as per **Table JSP.8-6** , "Variable synchronization behavior".

Table JSP.8-6 *Variable synchronization behavior*

	AT_BEGIN	NESTED	AT_END
Beginning of tag file	do nothing	save	do nothing
Before any fragment	*tag → page*	*tag → page*	do nothing
After any fragment	do nothing	do nothing	do nothing
End of tag file	*tag → page*	restore	*tag → page*

The following list describes what each synchronization action means. If `name-given` is used, the name of the variable in the calling page (referred to as P) and the name of the variable in the tag file (referred to as T) are the same and are equal to the value of `name-given`. If `name-from-attribute` is used, the name of P is equal to the value of the attribute (at the time the page was called) specified by the value of `name-from-attribute` and the name of T is equal to the value of the `alias` attribute.

- *tag → page* - For this variable, if T exists in the tag file, create/update P in the calling page. If a T does not exist in the tag file, and P does exist in the calling page, P is removed from the calling page's page scope. If the declare attribute for this variable is set to true , a corresponding scripting variable is declared in the calling page or tag file, as with any other tag handler. If this scripting variable would not be accessible in the context in which it is defined, the container need not declare the scripting variable (for example in a scriptless body).
- save - For this variable, save the value of P, for later restoration. If P did not exist, remember that fact.
- restore - For this variable, restore the value of P in the calling page, from the value saved earlier. If P did not exist before, ensure it does not exist now.

All variable synchronization and restoration that occurs at the end of a tag file must occur regardless of whether an exception is thrown inside the tag file. All variable synchronization that occurs after the invocation of a fragment must occur regardless of whether an exception occured while invoking the

fragment.

8.9.2. Synchronization Examples

The following examples help illustrate how variable synchronization works between a tag file and its calling page.

8.9.2.1. Example of AT_BEGIN

In this example, the `AT_BEGIN` scope is used to pass a variable to the tag's body, and make it available to the calling page at the end of the tag invocation.

```
<%-- page.jsp --%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="my" tagdir="/WEB-INF/tags" %>
<c:set var="x" value="1"/>
${x} <%-- (x == 1) --%>
<my:example>
  ${x} <%-- (x == 2) --%>
  <c:set var="x" value="3"/>
</my:example>
${x} <%-- (x == 4) --%>

<%-- /WEB-INF/tags/example.tag --%>
<%@ variable name-given="x" scope="AT_BEGIN" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
${x} <%-- (x == null) --%>
<c:set var="x" value="2"/>
<jsp:doBody/>
${x} <%-- (x == 2) --%>
<c:set var="x" value="4"/>
```

8.9.2.2. Example of AT_BEGIN and name-from-attribute

Like the previous example, in this example the `AT_BEGIN` scope is used to pass a variable to the tag's body, and make it available to the calling page at the end of the tag invocation. The name of the attribute is customized via `name-from-attribute`.

```
<%-- page.jsp --%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="my" tagdir="/WEB-INF/tags" %>
<c:set var="x" value="1"/>
${x} <%-- (x == 1) --%>
<my:example var="x">
  ${x} <%-- (x == 2) --%>
  ${result} <%-- (result == null) --%>
  <c:set var="x" value="3"/>
  <c:set var="result" value="invisible"/>
</my:example>
${x} <%-- (x == 4) --%>
${result} <%-- (result == 'invisible') --%>

<%-- /WEB-INF/tags/example.tag --%>
<%@ attribute name="var" required="true" rtexprvalue="false" %>
<%@ variable alias="result" name-from-attribute="var" scope="AT_BEGIN"
%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
${x} <%-- (x == null) --%>
${result} <%-- (result == null) --%>
<c:set var="x" value="ignored"/>
<c:set var="result" value="2"/>
<jsp:doBody/>
${x} <%-- (x == 'ignored') --%>
${result} <%-- (result == 2) --%>
<c:set var="x" value="still`ignored"/>
<c:set var="result" value="4"/>
```

8.9.2.3. Example of NESTED

In this example, the `NESTED` scope is used to make a private variable available to the calling page. The original value is restored when the tag is done.

```
<%-- page.jsp --%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="my" tagdir="/WEB-INF/tags" %>
<c:set var="x" value="1"/>
${x} <%-- (x == 1) --%>
<my:example>
  ${x} <%-- (x == 2) --%>
  <c:set var="x" value="3"/>
</my:example>
${x} <%-- (x == 1) --%>

<%-- /WEB-INF/tags/example.tag --%>
<%@ variable name-given="x" scope="NESTED" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
${x} <%-- (x == null) --%>
<c:set var="x" value="2"/>
<jsp:doBody/>
${x} <%-- (x == 2) --%>
<c:set var="x" value="4"/>
```

8.9.2.4. Example of AT_END

In this example, the AT_END scope is used to return a value to the page. The body of the tag is not affected.

```
<%-- page.jsp --%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="my" tagdir="/WEB-INF/tags" %>
<c:set var="x" value="1"/>
${x} <%-- (x == 1) --%>
<my:example>
  ${x} <%-- (x == 1) --%>
  <c:set var="x" value="3"/>
</my:example>
${x} <%-- (x == 4) --%>

<%-- /WEB-INF/tags/example.tag --%>
<%@ variable name-given="x" scope="AT_END" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
${x} <%-- (x == null) --%>
<c:set var="x" value="2"/>
<jsp:doBody/>
${x} <%-- (x == 2) --%>
<c:set var="x" value="4"/>
```

8.9.2.5. Example of Removing Parameters

This example illustrates how the tag file can remove objects from the page scope of the calling page during synchronization.

```
<%-- page.jsp --%>
<%@ taglib prefix="my" tagdir="/WEB-INF/tags" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<c:set var="x" value="2"/>
${x}
<my:tag1>
  '${x}'
</my:tag1>
${x}

<%-- /WEB-INF/tags/example.tag --%>
<%@ variable name-given="x" scope="NESTED" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<c:set var="x" value="1"/>
<jsp:doBody/>
<c:remove var="x"/>
<jsp:doBody/>
```

The expected output of this example is: 2 '1' '' 2

Chapter 9. Scripting

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

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

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

9.1. Overall Structure

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

9.1.1. Valid JSP Page

A JSP page is valid for a Java Platform if and only if the JSP page implementation class defined by **Table JSP.9-1** , "Structure of the Java Programming Language Class" (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.

9.1.2. Reserved Names

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

9.1.3. Implementation Flexibility

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

Table JSP.9-1 *Structure of the Java Programming Language Class*

Optional imports clause as indicated via `jsp` directive	`import name1`

`SuperClass` is either selected by the JSP container or by the JSP author via the `jsp` directive. Name of class (`jspXXX`) is implementation dependent.	`class jspXXX extends SuperClass`
Start of the body of a JSP page implementation class	`{`
(1) Declaration Section	`// declarations...`
signature for generated method	`public void _jspService(<ServletRequestSubtype> request, <ServletResponseSubtype> response) throws ServletException, IOException {`
(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`	`}`

9.2. Declarations Section

The declarations section corresponds to the declaration elements.

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

9.3. Initialization Section

This section defines and initializes the implicit objects available to the JSP page. See Section 1.8.3, “Implicit Objects”.

9.4. Main Section

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

The content of the main section 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.

9.4.1. Template Data

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

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

Original	Equivalent Text
`template`	`out.print(template)`

9.4.2. Scriptlets

A scriptlet is transformed into its code fragment.:

Original	Equivalent Text
`<% fragment %>`	`fragment`

9.4.3. Expressions

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

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

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

9.4.4. Actions

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

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

Original	Equivalent Text

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

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

Chapter 10. XML View

This chapter provides details on the XML view of a JSP page and tag files. The XML views are used to enable validation of JSP pages and tag files.

10.1. XML View of a JSP Document, JSP Page or Tag File

This section describes the XML view of a JSP page or tag file: the mapping between a JSP page, JSP document or tag file, and an XML document describing it.

10.1.1. JSP Documents and Tag Files in XML Syntax

The XML view of a JSP document or of a tag file written in XML syntax is very close to the original JSP page. Only five transformations are performed:

- Expand all include directives into the JSP content they include. See Section 1.10.5, “Including Data in JSP Pages” for the semantics of mixing XML and standard syntax content.
- Add a `jsp:root` element as the root element if the JSP document or tag file in XML syntax does not have it.
- Set the value of the `pageEncoding` attribute of the page directive to “UTF-8”. The `page` directive and the `pageEncoding` attribute are added if they don’t exist already.
- Set the value of the `contentType` attribute of the `page` directive to the value that the container will pass to `ServletResponse.setContentType()`, determined as described in Section 4.2, “Response Character Encoding”. The `page` directive and the `contentType` attribute are added if they don’t exist already.
- Add the `jsp:id` attribute (see Section 10.1.13, “The jsp:id Attribute”).

10.1.2. JSP Pages or Tag Files in JSP Syntax

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

- Expand all include directives into the JSP content they include. See Section 1.10.5, “Including Data in JSP Pages” for the semantics of mixing XML and standard syntax content.
- Add a `jsp:root` element as the root, with appropriate `xmlns:jsp` attribute, and convert the `taglib` directive into `xmlns:` attributes of the `jsp:root` element.
- Convert declarations, scriptlets, and expressions into valid XML elements as described in Section 6.3.2, “The jsp:root Element” and the following sections.
- Convert request-time attribute expressions as in Section 10.1.11, “Request-Time Attribute Expressions”.
- Convert JSP quotations to XML quotations.

- Create `jsp:text` elements for all template text.
- Add the `jsp:id` attribute (see Section 10.1.13, "The jsp:id Attribute").

Note that the XML view of a JSP page or tag file has no `DOCTYPE` information; see Section 10.2, "Validating an XML View of a JSP page".

A quick overview of the transformation is shown in **Table JSP.10-1** , "XML View Transformations":

Table JSP.10-1 *XML View Transformations*

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

In more detail:

10.1.3. JSP Comments

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

10.1.4. The page Directive

A `page` directive of the form:

```
<%@ page { attr="value" }* %>
```

is translated into an element of the form:

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

The value of the `pageEncoding` attribute is set to "UTF-8". The value of the `contentType` attribute is set to the value that the container will pass to ServletResponse.setContentType(), determined as described in Section 4.2, "Response Character Encoding". The `page` directive and both attributes are added if they don't exist already.

10.1.5. The taglib Directive

A taglib directive of the form

```
<%@ taglib uri="uriValue" prefix="prefix" %>
```

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

A taglib directive of the form

```
<%@ taglib tagdir="tagDirValue" prefix="prefix" %>
```

is translated into an `xmlns:prefix` attribute on the root of the JSP document, with a value of the form `urn:jsptagdir: tagDirValue`.

10.1.6. The include Directive

An include directive of the form

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

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

10.1.7. Declarations

Declarations are translated into a `jsp:declaration` element. For example, the second example from Section 1.12.1, "Declarations":

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

is translated into the following.

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

Alternatively, we could use an `<` and instead say:

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

10.1.8. Scriptlets

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

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

10.1.9. Expressions

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

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

10.1.10. Standard and Custom Actions

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

10.1.11. Request-Time Attribute Expressions

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

Request-time attribute values can also be specified using EL expressions of the form `${expression}`. Expressions of this form are represented verbatim in the XML view.

The XML view of an escaped EL expression using the ${expr} syntax can be obtained as follows:

- The XML view of an unescaped expression `${foo}` is `${foo}`.
- The XML view of an escaped expression `\${foo}` is `\${foo}`.

- For each escaped `\` preceeding an unescaped expression `${foo}`, a `${'\\'}` must be generated in the XML view, and neighboring generated `${'\\'}` expressions must be combined.

Table JSP.10-2 , "XML View of an Escaped EL Expression in a Request-time Attribute Value" illustrates these rules. Assume the EL expression `${foo}` evaluates to `[bar]` and that EL is enabled for this translation unit.

Table JSP.10-2 *XML View of an Escaped EL Expression in a Request-time Attribute Value*

Attribute Value	XML View	Result
`${foo}`	`${foo}`	`[bar]`
`\${foo}`	`\${foo}`	`${foo}`
`\\${foo}`	`${'\\'}${foo}`	`\[bar]`
`\\\${foo}`	`\\${foo}`	`\${foo}`
`\\\\${foo}`	`${'\\\\'}${foo}`	`\\[bar]`
`\\\\\${foo}`	`\\\${foo}`	`\\${foo}`
`\\\\\\${foo}`	`${'\\\\\\'}${foo}`	`\\\[bar]`
...	...	...

The XML view of an escaped EL expression using the `#{expr}` syntax follows the same rules as the `${expr}` syntax, where `${` is simply substituted with `#{`.

10.1.12. Template Text and XML Elements

All text that is uninterpreted by the JSP translator is converted into the body for a `jsp:text` element. As a consequence no XML elements of the form described in Section 6.3.9, "Template Content" will appear in the XML view of a JSP page written in JSP syntax.

Because `\\` is not an escape sequence within template text in the standard syntax, no special transformation needs to be done to obtain the XML view of an escaped EL expression that appears in template text.

Table JSP.10-3 , "XML View of an Escaped EL Expression in Template Text" illustrates how the XML view of an escaped EL expression is obtained. Assume the EL expression `${foo}` evaluates to `[bar]` and that EL is enabled for this translation unit. The same rules apply for the `#{expr}` syntax, where `${` is simply substituted with `\#{`.

Table JSP.10-3 *XML View of an Escaped EL Expression in Template Text*

Attribute Value	XML View	Result
`${foo}`	`${foo}`	`[bar]`
`\${foo}`	`\${foo}`	`${foo}`
`\\${foo}`	`\\${foo}`	`\${foo}`
`\\\${foo}`	`\\\${foo}`	`\\${foo}`

Attribute Value	XML View	Result
...	...	...

10.1.13. The jsp:id Attribute

A JSP container must support a `jsp:id` attribute. This attribute can only be present in the XML view of a JSP page and can be used to improve the quality of translation time error messages.

The XML view of any JSP page will have an additional `jsp:id` attribute added to all XML elements. This attribute is given a value that is unique over all elements in the XML view. The prefix for the `id` attribute need not be "jsp" but it must map to the namespace `http://java.sun.com/JSP/Page`. In the case where the page author has redefined the `jsp` prefix, an alternative prefix must be used by the container. See the `jakarta.servlet.jsp.tagext` Javadoc for more details.

10.1.14. The tag Directive

The `tag` directive is applicable to tag files only. A `tag` directive of the form:

```
<%@ tag { attr="value" }* %>
```

is translated into an element of the form:

```
<jsp:directive.tag { attr="value" }* />
```

The value of the `pageEncoding` attribute is set to "UTF-8". A `tag` directive and the `pageEncoding` attribute are added if they don't exist already.

10.1.15. The attribute Directive

The `attribute` directive is applicable to tag files only. An `attribute` directive of the form:

```
<%@ attribute { attr="value" }* %>
```

is translated into an element of the form:

```
<jsp:directive.attribute { attr="value" }* />
```

10.1.16. The variable Directive

The `variable` directive is applicable to tag files only. A `variable` directive of the form:

```
<%@ variable { attr="value" }* %>
```

is translated into an element of the form:

```
<jsp:directive.variable { attr="value" }* />
```

10.2. Validating an XML View of a JSP page

The XML view of a JSP page is a namespace-aware document and it cannot be validated against a DTD except in the most simple cases. To reduce confusion and possible unintended performance consequences, the XML view of a JSP page will not include a `DOCTYPE`.

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

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

10.3. Examples

This section presents various examples of XML Views. The first shows a JSP page in XML syntax that includes XML fragments. The second shows a JSP page in JSP syntax and its mapping to XML syntax. The three following examples illustrate the semantics of cross-syntax translation-time includes and the effect on the XML View.

10.3.1. A JSP Document

This is an example of a very simple JSP document that has some template XML elements. This particular example describes a table that is a collection of 3 rows, with numeric values 1, 2, 3. The JSP Standard Tag Library is being used:

```
<?xml version="1.0"?>
<table>
  <c:forEach
      xmlns:c="http://java.sun.com/jsp/jstl/core"
      var="counter" begin="1" end="3">
    <row>${counter}</row>
  </c:forEach>
</table>
```

10.3.2. A JSP Page and its Corresponding XML View

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

For this JSP page:

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

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

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

The XML View of the previous page is:

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

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




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

10.3.3. Clearing Out Default Namespace on Include

This example illustrates the need to clear out the default namespace when doing a translation-time include of a JSP document:

```
<!-- a.jspx -->
<elementA>
  <tagB xmlns="http://namespace1">
    <jsp:directive.include file="b.jspx"
        xmlns:jsp="http://java.sun.com/JSP/Page" />
  </tagB>
</elementA>

<!-- b.jspx -->
<elementC />
```

The resulting XML View for these two JSP documents is:

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page">
  <elementA>
    <tagB xmlns="http://namespace1">
      <elementC />
    </tagB>
  </elementA>
</jsp:root>
```

10.3.4. Taglib Direcive Adds to Global Namespace

This example illustrates the effect of the taglib directive on the XML View. Notice how the taglib directive always affects the `<jsp:root>` element, independent of where it is encountered.

```
<!-- c.jspx -->
<elementD xmlns:jsp="http://java.sun.com/JSP/Page">
  <jsp:directive.include file="d.jsp" />
  <jsp:directive.include file="e.jsp" />
</elementD>

<%-- d.jsp --%>
<%@ taglib prefix="x" uri="http://namespace2" %>
<x:tagE />

<%-- e.jsp --%>
<x:tagE />
```

The resulting XML View of these documents and pages is:

```
<jsp:root xmlns:x="http://namespace2"
    xmlns:jsp="http://java.sun.com/JSP/Page" >
  <elementD>
    <x:tagE />
    <x:tagE />
  </elementD>
</jsp:root>
```

10.3.5. Collective Application of Inclusion Semantics

This example illustrates how the various translation-time include semantics are collectively applied:

```
<%-- f.jsp --%>
<%@ taglib prefix="m" uri="http://namespace3" %>
<%@ include file="g.jspx" %>

<!-- g.jspx -->
<tagF xmlns="http://namespace4" >
  <y:tagG xmlns:y="http://namespace5">
    <tagH />
    <jsp:directive.include file="i.jspx"
        xmlns:jsp="http://java.sun.com/JSP/Page" />
  </y:tagG>
  <jsp:directive.include file="h.jsp"
      xmlns:jsp="http://java.sun.com/JSP/Page" />
  <tagI />
</tagF>

<%-- h.jsp --%>
<%@ taglib prefix="n" uri="http://namespace6" %>
<m:tagJ />
<n:tagK />

<!-- i.jspx -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page">
  <y:tagL xmlns:y="http://namespace7">
    <elementM />
    <jsp:directive.include file="h.jsp" />
  </y:tagL>
</jsp:root>
```

The resulting XML View of these documents and pages is:

```
<jsp:root xmlns:m="http://namespace3"
    xmlns:n="http://namespace6"
    xmlns:jsp="http://java.sun.com/JSP/Page" >
  <tagF xmlns="http://namespace4">
    <y:tagG xmlns:y="http://namespace5">
      <tagH />
      <y:tagL xmlns="" xmlns:y="http://namespace7">
        <elementM />
        <m:tagJ />
        <n:tagK />
      </y:tagL>
    </y:tagG>
    <m:tagJ />
    <n:tagK />
    <tagI />
  </tagF>
</jsp:root>
```

Part II

The next chapter provides detail specification information on some portions of the JSP specification that are intended for JSP Container Vendors, JSP Page authors, and JSP Tag Library authors.

The chapters is normative.

The chapter is:

- JSP Container

JSP Container Vendors, JSP Page authors, and JSP Tag Library authors should also read:

- `jakarta.servlet.jsp` Javadoc
- `jakarta.servlet.jsp.tagext` Javadoc
- Expression Language Specification

These external resources are considered normative within the context of this specification.

Chapter 11. JSP Container

This chapter describes the contracts between a JSP container and a JSP page, including the precompilation protocol and debugging support requirements.

The information in this chapter is independent of the Scripting Language used in the JSP page. Chapter 9, *Scripting* 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.

11.1. JSP Page Model

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

The JSP page describes how to create a response object from a request object for a given protocol, possibly creating and/or using some other objects in the process . A JSP page may also indicate how some events are to be handled. In JSP 3.0 only `init` and `destroy` events are allowed events.

The JSP container must render a JSP page for the HTTP methods GET and POST with identical responses. The response for a HEAD request to a JSP page must be identical to the response for a GET request minus the response body. The behavior of the JSP container is undefined for other methods.

11.1.1. Protocol Seen by the Web Server

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

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

The JSP container automatically makes a number of server-side objects available to the JSP page implementation object . See Section 1.8.3, “Implicit Objects”.

11.1.1.1. Protocol Seen by the JSP Page Author

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

The main portion of this contract is the `_jspService` method that is generated automatically by the JSP

container from the JSP page. The details of this contract are provided in Chapter 9, *Scripting*.

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 3.0 this is done by defining methods with the names `jspInit` and `jspDestroy` in a declaration scripting element in the JSP page. The `jspInit` method, if present, will be called to prepare the page before the first request is delivered. Similarly a JSP container can reclaim resources used by a JSP page when a request is not being serviced by the JSP page by invoking its `jspDestroy` method, if present.

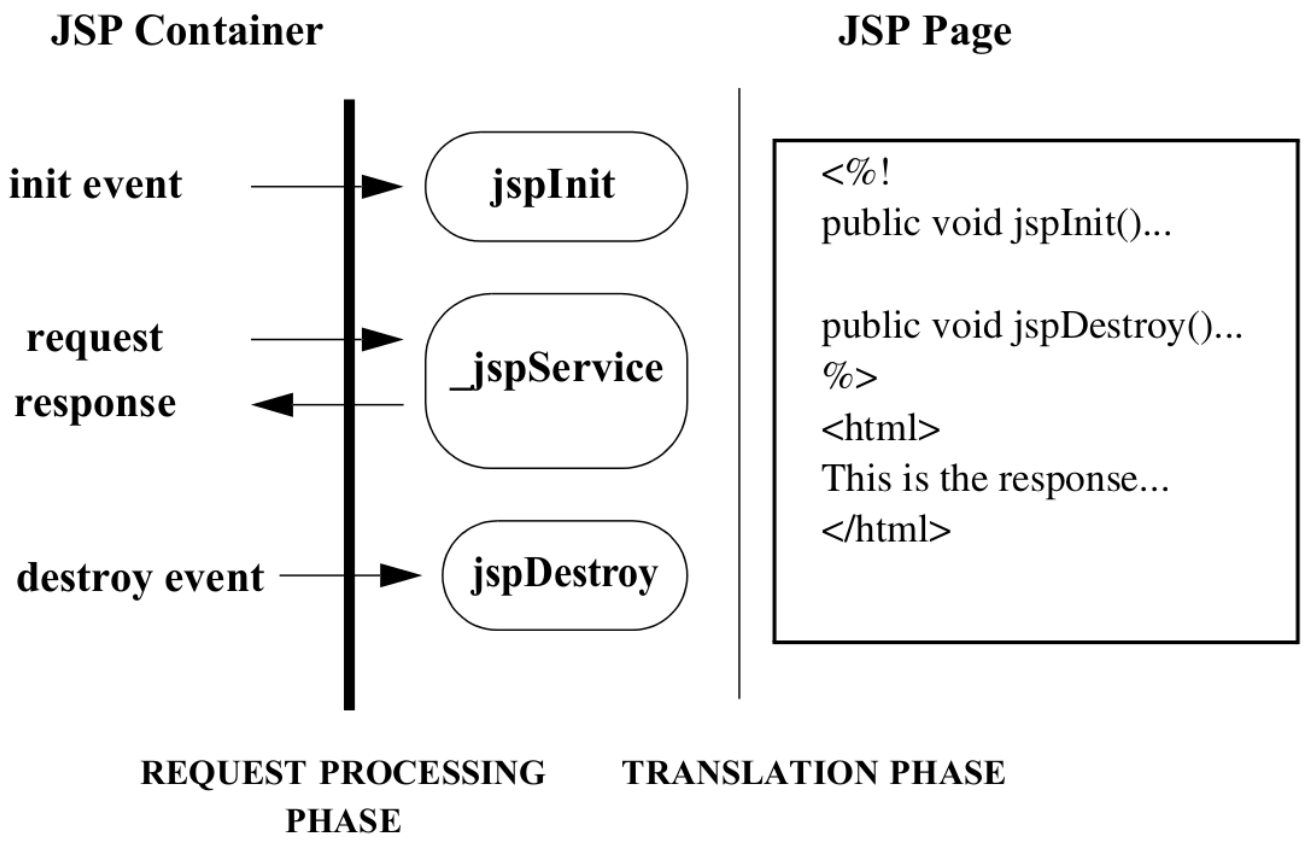
A JSP page author may not (re)define servlet methods through a declaration scripting element.

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

11.1.1.2. The HttpJspPage Interface

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

Figure JSP.11-1 *Contracts between a JSP Page and a JSP Container*



The involved contracts are shown in **Figure JSP.11-1** Contracts between a JSP Page and a JSP Container. We now revisit this whole process in more detail.

11.2. JSP Page Implementation Class

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

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

The JSP Page implementation object belongs to an implementation-dependent named package. The package used may vary between one JSP and another, so minimal assumptions should be made.

As of JSP 2.0, it is illegal to refer to any classes from the unnamed (a.k.a. default) package. This will result in a translation error. This restriction also applies for all other cases where classes are referenced, such as when specifying the class name for a tag in a TLD.

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

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

The JSP page implementation class will implement `jakarta.servlet.Servlet` and requests are delivered to the class as per the rules in the Servlet 5.0 specification.

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

A JSP page author writes a JSP page expecting that the client and the server will communicate using a certain protocol. The JSP container must guarantee that requests to and responses from the page use that protocol. Most JSP pages use HTTP, and their implementation classes must implement the `HttpJspPage` interface, which extends `JspPage`. If the protocol is not HTTP, then the class will implement an interface that extends `JspPage`.

11.2.1. API Contracts

The contract between the JSP container and a Java class implementing a JSP page corresponds to the `Servlet` interface. Refer to the Servlet 5.0 specification for details.

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

Table JSP.11-1 *How the JSP Container Processes JSP Pages*

Methods the JSP Container Invokes	Comments
`void jspInit()`	Method is optionally defined in JSP page. Method is invoked when the JSP page is initialized. When method is called all the methods in `servlet`, including `getServletConfig` are available.
`void jspDestroy()`	Method is optionally defined in JSP page. Method is invoked before destroying the page.
`void _jspService(<ServletRequestSubtype>, <ServletResponseSubtype>) throws IOException, ServletException`	Method may **not** be defined in JSP page. The JSP container automatically generates this method, based on the contents of the JSP page. Method invoked at each client request.

11.2.2. Request and Response Parameters

As shown in **Table JSP.11-1** , "How the JSP Container Processes JSP Pages" 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 `jakarta.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 `jakarta.servlet.ServletResponse`. The interface must define a protocol-dependent response contract between the JSP container and the class that implements the JSP page.

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

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

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

11.2.3. Omitting the extends Attribute

If the `extends` attribute of the `page` directive (see Section Section 1.10.1, “The `page` Directive”) in a JSP page is not used, the JSP container can generate any class that satisfies the contract described in **Table JSP.11-1** , “How the JSP Container Processes JSP Pages” when it transforms the JSP page.

In the following code examples, **Code Example JSP.11-1** A Generic HTTP Superclass illustrates a generic HTTP superclass named `ExampleHttpSuper`. **Code Example JSP.11-2** The Java Class Generated From a JSP Page shows a subclass named `_jsp1344` that extends `ExampleHttpSuper` and is the class generated from the JSP page. By using separate `_jsp1344` and `ExampleHttpSuper` classes, the JSP page translator does not need to discover whether the JSP page includes a declaration with `jspInit` or `jspDestroy`. This significantly simplifies the implementation.

Code Example JSP.11-1 *A Generic HTTP Superclass*

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

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

abstract class ExampleHttpSuper implements HttpJspPage {
  private ServletConfig config;

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

  public void jspInit() {
  }

  public void jspDestroy() {
  }

  final public ServletConfig getServletConfig() {
    return config;
  }

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

  final public void destroy() {
```

```
    jspDestroy();
  }

  /**
   * The entry point into service.
   */
  final public void service(ServletRequest req, ServletResponse res)
      throws ServletException, IOException {

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

    _jspService(request, response);
  }

  /**
   * abstract method to be provided by the JSP processor in the subclass
   * Must be defined in subclass.
   */
  abstract public void _jspService(HttpServletRequest request,
      HttpServletResponse response) throws ServletException, IOException;
  }
```

Code Example JSP.11-2 *The Java Class Generated From a JSP Page*

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

/**
 * An example of a class generated for a JSP.
 *
 * The name of the class is unpredictable.
 * We are assuming that this is an HTTP JSP page (like almost all are)
 */
class _jsp1344 extends ExampleHttpSuper {

  // Next code inserted directly via declarations.
  // Any of the following pieces may or not be present
  // if they are not defined here the superclass methods
  // will be used.

  public void jspInit() {....}
  public void jspDestroy() {....}

  // The next method is generated automatically by the
  // JSP processor.
  // body of JSP page

  public void _jspService(HttpServletRequest request,
      HttpServletResponse response)
      throws ServletException, IOException {

    // initialization of the implicit variables
    // ...

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

  }
}
```

11.2.4. Using the extends Attribute

If the JSP page author uses `extends`, the generated class is identical to the one shown in **Code Example JSP.11-2** The Java Class Generated From a JSP Page, except that the class name is the one specified in the `extends` attribute.

The contract on the JSP page implementation class does not change. The JSP container should check (usually through reflection) that the provided superclass:

- Implements `HttpJspPage` if the protocol is HTTP, or `JspPage` otherwise.
- All of the methods in the `Servlet` interface are declared final.

Additionally, it is the responsibility of the JSP page author that the provided superclass satisfies:

- The `service` method of the servlet API invokes the `_jspService` method.
- The `init(ServletConfig)` method stores the configuration, makes it available via `getServletConfig`, then invokes `jspInit`.
- The `destroy` method invokes `jspDestroy`.

A JSP container may give a fatal translation error if it detects that the provided superclass does not satisfy these requirements, but most JSP containers will not check them.

11.3. Buffering

The JSP container buffers data (if the jsp directive specifies it using the buffer attribute) as it is sent from the server to the client. Headers are not sent to the client until the first flush method is invoked. Therefore, it is possible to call methods that modify the response header, such as `setContentType`, `sendRedirect`, or error methods, up until the flush method is executed and the headers are sent. After that point, these methods become invalid, as per the Servlet specification.

The `jakarta.servlet.jsp.JspWriter` class buffers and sends output. The `JspWriter` class is used in the `_jspService` method as in the following example:

```
import jakarta.servlet.jsp.JspWriter;

static JspFactory jspFactory = JspFactory.getDefaultFactory();

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

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

The complete listing of `jakarta.servlet.jsp.JspWriter` can be found in the `jakarta.servlet.jsp` Javadoc.

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

11.4. Precompilation

A JSP page that is using the HTTP protocol will receive HTTP requests. JSP 3.0 compliant containers must support a simple precompilation protocol, as well as some basic reserved parameter names. Note that the precompilation protocol is related but not the same as the notion of compiling a JSP page into a `Servlet` class (Appendix A, *Packaging JSP Pages*).

11.4.1. Request Parameter Names

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

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

11.4.2. Precompilation Protocol

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

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

For example:

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

1, 2, and 4 are legal; the request will not be delivered to the page. 3 and 5 are legal; the request will not be delivered to the page.

6. `?jsp_precompile=foo`

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

11.5. Debugging Requirements

The Jakarta Debugging Support for Other Languages specification provides the JSP Compiler with a standard format to convey source map debugging information to tools such as debuggers. See https://jakarta.ee/specifications/debugging/2.0/ for details.

JSP 3.0 containers are required to provide debugging support for JSP pages and tag files written in either standard or XML syntax.

The JSP compiler must produce `.class` files with a `SourceDebugExtension` attribute, mapping each line or lines of JSP code to the corresponding generated line or lines of Java code. For both pages and tag files, the stratum that maps to the original source should be named `JSP` in the Source Debug Extension (this stratum name is reserved for use by the JSP specification). This stratum should be specified as the default, unless the page or tag file was generated from some other source.

The exact mechanism for causing the JSP compiler to produce source map debugging information is implementation-dependent.

11.5.1. Line Number Mapping Guidelines

The following is a set of non-normative guidelines for generating high quality line number mappings. The guidelines are presented to help produce a consistent debugging experience for page authors across containers. Where possible the JSP container should generate line number mappings as follows:

1. A breakpoint on a JSP line causes execution to stop before any Java code which amounts to a translation of the JSP line is executed (for one possible exception, see (5). Note that given the LineInfo Composition Algorithm (see Jakarta Debugging Support for Other Languages specification), it is acceptable for the mappings to include one or more Java lines which are never translated into executable byte code, as long as at least one of them does.
2. It is permitted for two or more lines of JSP to include the same Java lines in their mappings.
3. If a line of JSP has no manifestation in the Java source other than white-space preserving source, it should not be mapped.
    - The following standard syntax JSP entities should not be mapped to generated code. These entities either have no manifestation in the generated Java code (e.g. comments), or are not manifest in such a way that it allows the debugged process to stop (e.g. the page directive import):
        - JSP comments
        - Directives
    - The following XML syntax JSP entities should not be mapped to generated code. These entities frequently have no manifestation in the generated Java code.
        - `<jsp:root>`
        - `<jsp:output>`

4. Declarations and scriptlets (standard or XML JSP). Lines in these constructs should preserve a one-to-one mapping with the corresponding generated code lines. Empty lines and comment lines are not mapped.
5. For scriptlets, scriptlet expressions, EL expressions, standard actions and custom actions in template text, a line containing one or more of these entities should be mapped to Java source lines which include the corresponding Java code.

   If the line starts with template text, the Java code which handles it may be excluded from the mappings if this would cause the debugger to stop before the apparent execution of JSP lines preceding the line in question. For example:

   ```
   100 <p>This is a line with template text.</p>
   101 <h1><fmt:message key="company" bundle="${bundle}"/></h1>

   200 out.write( "<p>This is a line with template text.</p>\r\n" );
   201 out.write( "<h1>" );
   202 org.apache.taglibs.standard.tag.el.fmt.MessageTag taghandler =
   203 new org.apache.taglibs.standard.tag.el.fmt.MessageTag();
   204 taghandler.setPageContext( pageContext );
   205 ...
   ```

   In this example, given that `<h1>` has its own call to `write()`, it makes sense to map 101 to 201, 202 etc.

   ```
   200 out.write( "<p>This is a line with template text.</p>\r\n<h1>" );
   201 org.apache.taglibs.standard.tag.el.fmt.MessageTag taghandler =
   202 new org.apache.taglibs.standard.tag.el.fmt.MessageTag();
   203 taghandler.setPageContext( pageContext );
   204 ...
   ```

   In this second example, given that `<h1>` is output using the same call to `write()` that was used for line 100, mapping 101 to 202, 203 etc. may result in more intuitive behavior of the debugger.

   For scriptlets that contain more than one line, there should be a one-to-one mapping from JSP to Java lines, and the mapping should start at the first Java code that is not whitespace or comments. Therefore, a line that contains only the open scriptlet delimeter is not mapped.

6. Scriptlet expressions and EL expressions in attribute values. The source line mappings should include any Java source lines that deal with the evaluation of the rtexpr value as well as source that deals with the JSP action.
7. Standard or custom actions.
   - Empty tags and start tags special case: The `jsp:params` action typically has no manifestation and should not be mapped.

- Empty tags and start tags: The Java line mappings should include as much of the corresponding Java code as possible, including any separate lines that deal with rtexpr evaluation as described in (6). If it is not possible to include all the Java code in the mappings, the mapped lines should include the first sequential line which deals with either the tag or the attribute evaluation in order to meet (1)
- Closing tags frequently do not have a manifestation in the Java source, but sometimes do. In case a JSP line contains only a closing tag, the line may be mapped to whitespace preserving Java source if it has no semantic translation. This will avoid a confusing user experience where it is sometimes possible to set a breakpoint on a line consisting of a closing tag and sometimes not.

Part III

Appendices B is normative. Appendices A, C, and D are non-normative.

The Appendices are:

- Appendix A - Packaging JSP pages
- Appendix B - Page Character Encoding Detection Algorithm
- Appendix C - Changes
- Appendix D - Glossary of terms

Appendix A: Packaging JSP Pages

This appendix shows two simple examples of packaging a JSP page into a WAR for delivery into a Web container. In the first example, the JSP page is delivered in source form. This is likely to be the most common example. In the second example the JSP page is compiled into a servlet that uses only Servlet 5.0 and JSP 3.0 API calls; the servlet is then packaged into a WAR with a deployment descriptor such that it looks as the original JSP page to any client.

This appendix is non normative. Actually, strictly speaking, the appendix relates more to the Servlet 5.0 capabilities than to the JSP 3.0 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 `com.acme._jsp_HelloWorld_XXX_Impl`. The servlet code only depends on the JSP 3.0 and Servlet 5.0 APIs, as follows:

```
package com.acme;

import jakarta.servlet.*;
import jakarta.servlet.http.*;
import jakarta.servlet.jsp.*;

public class _jsp_HelloWorld_XXX_Impl
    extends PlatformDependent_Jsp_Super_Impl {

  public void jspInit() {
    // ...
  }

  public void jspDestroy() {
    // ...
  }

  static JspFactory _factory = JspFactory.getDefaultFactory();

  public void _jspService(HttpServletRequest request,
      HttpServletResponse response)
      throws IOException, ServletException {

    Object page = this;
    HttpSession session = request.getSession();
    ServletConfig config = getServletConfig();
    ServletContext application = config.getServletContext();

    PageContext pageContext = _factory.getPageContext(this,
        request,
        response,
        (String)NULL,
        true,
        JspWriter.DEFAULT_BUFFER,
        true);

    JspWriter out = 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>com.acme._jsp_HelloWorld_XXX_Impl</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>HelloWorld</servlet-name>
    <url-pattern>/HelloWorld.jsp</url-pattern>
  </servlet-mapping>

  <session-config>
    <session-timeout>1</session-timeout>
   </session-config>
</webapp>
```

A.5. The WAR for the Compiled JSP Page

Finally everything is packaged together into a WAR:

- `/WEB-INF/web.xml`
- `/WEB-INF/classes/com/acme/_jsp_HelloWorld_XXX_Impl.class`

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

Appendix B: Page Encoding Detection

This appendix details the algorithm containers are required to use in order to determine the character encoding for a JSP page or tag file. See Chapter 4, *Internationalization Issues* for details on where this algorithm is used. The algorithm is designed to maximize convenience to the page author, while preserving backwards compatibility with previous versions of the JSP specification.

B.1. Detection Algorithm for JSP pages

The following is a complete though unoptimized algorithm for determining the character encoding for a JSP file. JSP containers may use an optimized version of this algorithm, but it must detect the same encoding as the algorithm in all cases.

1. Decide whether the source file is a JSP page in standard syntax or a JSP document in XML syntax.
   a. If there is a `<is-xml>` element in a `<jsp-property-group>` that names this file, then if it has the value “true”, the file is a JSP document, and if it has the value “false”, the file is not a JSP document.
   b. Otherwise, if the file name has the extension “jspx”, the file is a JSP document.
   c. Otherwise, try to find a `<jsp:root>` element in the file.
      i. Determine the initial encoding from the first four bytes of the file, as described in appendix F.1 of the XML 1.0 specification. For the byte sequence “3C 3F 78 6D”, use `ISO-8859-1` ; for the byte sequence “4C 6F A7 94”, use `IBM037` ; for all other cases, use the `UTF- or UCS-` encoding given in the appendix.
      ii. Read the file using the initial encoding and search for a `<jsp:root>` element. If the element is found and is the top element, the file is a JSP document in XML syntax
   d. Otherwise, the file is a JSP page in standard syntax.
2. Reset the file.
3. If the file is a JSP page in standard syntax:

   3.1 If the file is not preceded by a BOM:

   a. Check whether there is a JSP configuration element `<page-encoding>` whose URL pattern matches this file.
   b. Read the file using the initial encoding and search for a `pageEncoding` attribute in a page declaration. The specification requires the attribute to be found only if it is not preceded by non-ASCII characters, so simplified implementations are allowed.
   c. Report an error if there are a `<page-encoding>` configuration element whose URL pattern matches this file and a `pageEncoding` attribute, and the two name different encodings.
   d. If there is a `<page-encoding>` configuration element whose URL pattern matches this file, the page character encoding is the one named in this element.

e. Otherwise, if there is a `pageEncoding` attribute, the page character encoding is the one named in this attribute.

f. Otherwise, read the file using the initial encoding and search for a charset value within a `contentType` attribute in a page declaration. If it exists, the page character encoding is the one named in this charset value. The specification requires the attribute to be found only if it is not preceded by non-ASCII characters, so simplified implementations are allowed.

g. Otherwise, the page character encoding is `ISO-8859-1`.

3.2 If the file is preceded by a BOM:

h. Read the file using the encoding indicated by the BOM, and search for a pageEncoding attribute in a page declaration.

i. Report an error if any of the following conditions are met:

   i. There is a <page-encoding> configuration element whose URL pattern matches this page and whose encoding does not match the encoding indicated by the BOM.

   ii. There is a pageEncoding page directive attribute whose encoding does not match the encoding indicated by the BOM.

4. If the file is a JSP document in XML syntax, use these steps.

   a. Determine the page character encoding as described in appendix F.1 of the XML 1.0 specification. Note whether the encoding was named in the encoding attribute of the XML prolog or just derived from the initial bytes.

   b. Check whether there is a JSP configuration element `<page-encoding>` whose URL pattern matches this file.

   c. Read the file using the detected encoding and search for a `pageEncoding` attribute in a `<jsp:directive.page>` element.

   d. Report an error if any of the following conditions is met:

      i. The XML prolog names an encoding and there is `<page-encoding>` configuration element whose URL pattern matches this file and which names a different encoding.

      ii. The XML prolog names an encoding and there is a `pageEncoding` attribute which names a different encoding.

      iii. There are a `<page-encoding>` configuration element whose URL pattern matches this file and a `pageEncoding` attribute, and the two name different encodings.

5. Reset the file and read it using the page character encoding.

B.2. Detection Algorithm for Tag Files

The following details the algorithm for determining the character encoding for a tag file. JSP containers may use an optimized version of this algorithm, but it must detect the same encoding as the algorithm in all cases.

1. Determine whether the source file is a tag file in standard or XML syntax.
   a. If the file name has the extension "tagx", the file is a tag file in XML syntax. Otherwise, it is a tag file in standard syntax.
2. If the file is a tag file in standard syntax, use these steps:

   2.1 If the file is not preceded by a BOM:

   a. Read the file using the initial default encoding and search for a `pageEncoding` attribute in a tag directive. The specification requires the attribute to be found only if it is not preceded by non-ASCII characters.
   b. If there is a `pageEncoding` attribute, the page character encoding is the one named in this attribute.
   c. Otherwise, the page character encoding is `ISO-8859-1`.
   d. Reset the file and read it using the page character encoding.

   2.2 If the file is preceded by a BOM:

   e. Read the file using the encoding indicated by the BOM, and search for a pageEncoding attribute in a tag directive.
   f. Report an error if there is a pageEncoding tag directive attribute whose encoding does not match the encoding indicated by the BOM.
3. If the file is a JSP document in XML syntax, use these steps.
   a. Determine the page character encoding as described in appendix F.1 of the XML 1.0 specification.
   b. Read the file using the detected encoding.

Appendix C: Changes

This appendix lists the changes in the Jakarta Server Pages specification. This appendix is non-normative.

C.1. Changes between JSP 3.0 and JSR 245

- The API has moved from the `javax.servlet.jsp` package to the `jakarta.servlet.jsp` package.
- All deprecated methods now include the `@Deprecated` annotation.
- All API methods use generics where appropriate.
- The contents of the Javadoc and XML schemas were removed and are now included by reference.

Appendix D: Glossary

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

action
An element in a JSP page that can act on implicit objects and other server-side objects or can define new scripting variables. Actions follow the XML syntax for elements with a start tag, a body and an end tag; if the body is empty it can also use the empty tag syntax. The tag must use a prefix.

action, standard
An action that is defined in the JSP specification and is always available to a JSP file without being imported.

action, custom
An action described in a portable manner by a tag library descriptor and a collection of Java classes and imported into a JSP page by a taglib directive.

Application Assembler
A person that combines JSP pages, servlet classes, HTML content, tag libraries, and other Web content into a deployable Web application.

classic tag handler
A tag handler that implements the `jakarta.servlet.jsp.tagext.Tag` interface.

component contract
The contract between a component and its container, including life cycle management of the component and the APIs and protocols that the container must support.

Component Provider
A vendor that provides a component either as Java classes or as JSP page source.

distributed container
A JSP container that can run a Web application that is tagged as distributable and is spread across multiple Java virtual machines that might be running on different hosts.

declaration
A scripting element that declares methods, variables, or both in a JSP page. Syntactically it is delimited by the `<%!` and `%>` characters.

directive
An element in a JSP page that gives an instruction to the JSP container and is interpreted at translation time. Syntactically it is delimited by the `<%@` and `%>` characters.

dynamic attribute
An attribute, passed to a custom action, whose name is not explicitly declared in the tag library

descriptor.

element
A portion of a JSP page that is recognized by the JSP translator. An element can be a directive, an action, or a scripting element.

EL expression
An element in a JSP page representing an expression to be parsed and evaluated via the JSP Expression Language. Syntactically it is delimited by the `${` and `}` characters.

expression
Either a scripting expression or an EL expression.

fixed template data
Any portions of a JSP file that are not described in the JSP specification, such as HTML tags, XML tags, and text. The template data is returned to the client in the response or is processed by a component.

implicit object
A server-side object that is defined by the JSP container and is always available in a JSP file without being declared. The implicit objects are `request`, `response`, `pageContext`, `session`, `application`, `out`, `config`, `page`, and `exception` for scriptlets and scripting expressions. The implicit objects are `pageContext`, `pageScope`, `requestScope`, `sessionScope`, `applicationScope`, `param`, `paramValues`, `header`, `headerValues`, `cookie` and `initParam` for EL expressions.

Jakarta Server Pages technology
An extensible Web technology that uses template data, custom elements, scripting languages, and server-side Java objects to return dynamic content to a client. Typically the template data is HTML or XML elements, and in many cases the client is a Web browser.

JSP container
A system-level entity that provides life cycle management and runtime support for JSP and servlet components.

JSP configuration
The deployment-time process by which the JSP container is declaratively configured using a deployment descriptor.

JSP file
A text file that contains JSP elements, forming a complete JSP page or just a partial page that must be combined with other JSP files to form a complete page. Most top-level JSP files have a `.jsp` extension, but other extensions can be configured as well.

JSP fragment
A portion of JSP code, translated into an implementation of the `jakarta.servlet.jsp.JspFragment` abstract class.

JSP page

One or more JSP files that form a syntactically complete description for processing a request to create a response.

JSP page, front
A JSP page that receives an HTTP request directly from the client. It creates, updates, and/or accesses some server-side data and then forwards the request to a presentation JSP page.

JSP page, presentation
A JSP page that is intended for presentation purposes only. It accesses and/or updates some server-side data and incorporates fixed template data to create content that is sent to the client.

JSP page implementation class
The Java programming language class, a servlet, that is the runtime representation of a JSP page and which receives the request object and updates the response object. The page implementation class can use the services provided by the JSP container, including both the servlet and the JSP APIs.

JSP page implementation object
The instance of the JSP page implementation class that receives the `request` object and updates the `response` object.

JSP segment
A portion of JSP code defined in a separate file, and imported into a page using the include directive.

named attribute
A standard or custom action attribute whose value is defined using the `<jsp:attribute>` standard action.

scripting element
A declaration, scriptlet, or expression, whose tag syntax is defined by the JSP specification, and whose content is written according to the scripting language used in the JSP page. The JSP specification describes the syntax and semantics for the case where the language page attribute is `java`.

scripting expression
A scripting element that contains a valid scripting language expression that is evaluated, converted to a String, and placed into the implicit out object. Syntactically it is delimited by the `<%=` and `%>` characters.

scriptlet
An scripting element containing any code fragment that is valid in the scripting language used in the JSP page. The JSP specification describes what is a valid scriptlet for the case where the language page attribute is `java`. Syntactically a scriptlet is delimited by the `<%` and `%>` characters.

simple tag handler
A tag handler that implements the `jakarta.servlet.jsp.tagext.SimpleTag` interface.

tag
A piece of text between a left angle bracket and a right angle bracket that has a name, can have

attributes, and is part of an element in a JSP page. Tag names are known to the JSP translator, either because the name is part of the JSP specification (in the case of a standard action), or because it has been introduced using a Tag Library (in the case of custom action).

tag file
A text-based document that uses fixed template data and JSP elements to define a custom action. The semantics of a tag file are realized at runtime by a tag handler.

tag 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 Jakarta EE 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.