

# JavaServer Pages™ Specification

---

*Version 1.0 - PUBLIC DRAFT 1 -*

*please send feedback to:  
jsp-comments@calterra.eng.sun.com*



THE NETWORK IS THE COMPUTER™

Java Software  
A Division of Sun Microsystems, Inc.  
901 San Antonio Road  
Palo Alto, California 94303  
415 960-1300 fax 415 969-9131

April 29, 1999

---

Eduardo Pelegri-Llopart, Larry Cable  
with Suzanne Ahmed

## **JavaServer Pages™ Specification (“Specification”)**

**Version: 1.0 - PUBLIC DRAFT 1 -**

**Status: Evaluation Posting**

**Release: April 29, 1999**

Copyright 1998-99 Sun Microsystems, Inc.  
901 San Antonio Road, Palo Alto, California 94303, U.S.A.  
All rights reserved.

### **NOTICE**

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

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

### **TRADEMARKS**

No right, title, or interest in or to any trademarks, service marks, or trade names of Sun or Sun’s licensor is granted hereunder.

Sun, Sun Microsystems, the Sun logo, Java, JavaBeans, Enterprise JavaBeans, JavaServer, JavaServer Pages, JDK, JDBC, Java 2, The Network is the Computer, and Write Once, Run Anywhere are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

### **DISCLAIMER OF WARRANTIES**

THIS SPECIFICATION IS PROVIDED “AS IS” AND IS EXPERIMENTAL AND MAY CONTAIN DEFECTS OR DEFICIENCIES WHICH CANNOT OR WILL NOT BE CORRECTED BY SUN. SUN MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT; THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE; NOR THAT ANY PRACTICE OR IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE

ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADE SECRETS OR OTHER RIGHTS. This document does not represent any commitment to release or implement any portion of this Specification in any product(s).

THIS SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW VERSIONS OF THE SPECIFICATION, IF ANY. SUN MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS SPECIFICATION AT ANY TIME. Any use of such changes in the Specification will be governed by the then current terms and conditions for the applicable version of the Specification.

### **LIMITATION OF LIABILITY**

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

You will indemnify, hold harmless, and defend Sun and its licensors from any claims based on your use of the Specification for any purposes other than those of internal evaluation, and from any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

### **RESTRICTED RIGHTS LEGEND**

Use, duplication, or disclosure by the U.S. Government is subject to the restrictions set forth in this license and as provided in DFARS 227.7202-1(a) and 227.7202-3(a) (1995), DFARS 252.227-7013(c)(1)(ii)(Oct 1988), FAR 12.212(a) (1995), FAR 52.227-19 (June 1987), or FAR 52.227-14(ALT III) (June 1987), as applicable.

### **REPORT**

As an Evaluation Posting of this Specification, you may wish to report any ambiguities, inconsistencies, or inaccuracies you may find in connection with your evaluation of the Specification (“Feedback”). To the extent that you provide Sun with any Feedback, you hereby: (i) agree that that such Feedback is provided on a non-proprietary and non-confidential basis and (ii) grant to Sun a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license to incorporate, disclose, and use without limitation the Feedback for any purpose relating to the Specification and future versions, implementations, and test suites thereof.

# Status of this Specification

---

This is the JavaServer Pages™ (JSP) 1.0 specification - PUBLIC DRAFT 1 - . This specification builds on previously published specifications 0.91 and 0.92 and reflects a significant body of feedback and experience with those specifications. Additionally, in the last few months, the core JSP group at Sun Microsystems has made a significant effort to add a number of additional “participants” to the specification process. Although undoubtedly we have missed some key participants, we believe that the current specification reflects the input of all these parties.

We believe this version of the specification to be pretty close to the final JSP 1.0 specification but, as with any specification that is not yet final, we cannot promise that specific features will or not change. We expect an interim update to this draft in a few weeks based on feedback from all involved parties; we hope that update will include the last final changes.

Although an attempt has been made to preserve the spirit and general direction of previous public drafts, incorporating all the feedback has led to an specification that is not upwardly compatible with those drafts.

The JSP 1.0 specification will be followed by another, JSP 1.1, see Appendix D for more details.

## *Areas known to need extra work*

There are a few areas that are known to still need some work; we encourage your feedback.

1. Whether any optional features become mandatory.
2. Clarify, correct, and finalize PageContext issues.
3. Complete and correct Chapter 3 (XML syntax).
4. Define I18N properties of JSP 1.0, including treatment of content type.

5. Provide some guidance on class reloading.

# Contents

---

<b>Status of this Specification.....</b>	<b>iii</b>
<b>Preface .....</b>	<b>xi</b>
Who should read this document .....	xi
Related Documents .....	xi
Future Directions.....	xii
<b>Chapter 1: Overview .....</b>	<b>15</b>
Why JavaServer Pages?.....	15
What is a JSP Page? .....	16
Features in JSP 1.0.....	17
The JSP Model.....	18
Objects and Scopes .....	19
Fixed Template Data .....	20
Directives and Actions.....	20
Scripting Languages.....	21
Objects and Variables.....	22
JSP, HTML, and XML.....	22
A Web Application.....	23
Application Models.....	24

Simple 21/2-Tier Application.....	25
N-tier Application.....	25
Loosely Coupled Applications.....	26
Using XML with JSP .....	27
Redirecting Requests .....	28
Including Requests .....	29
<b>Chapter 2: Core Syntax and Semantics .....</b>	<b>31</b>
General Syntax Rules.....	31
JSP Elements and Template Data .....	31
JSP Element Syntax.....	32
Start and End Tags .....	32
Empty Elements .....	32
Attribute Values .....	32
White Space .....	33
Error Handling .....	34
Translation Time Processing Errors .....	34
Client Request Time Processing Errors .....	34
Unimplemented Optional Feature.....	35
Comments .....	35
Quoting and Escape Conventions.....	36
Web Applications.....	36
Relative URL Specifications within an Application ....	37
Overview of Semantics.....	37
Template Text Semantics.....	38
Directives .....	38
The page Directive .....	39
Including Data in JSP Pages. ....	45

The include Directive .....	45
The taglib Directive .....	46
Implicit Objects .....	47
The pageContext Object.....	49
Scripting Elements.....	49
Declarations .....	50
Scriptlets .....	50
Expressions .....	51
Actions .....	52
Tag Attribute Interpretation Semantics .....	53
Request Time Attribute Values .....	53
The id Attribute .....	54
The scope Attribute .....	55
Standard Actions .....	56
<jsp:useBean> .....	56
<jsp:setProperty> .....	58
<jsp:getProperty> .....	60
<jsp:include>.....	61
<jsp:request> .....	62
<jsp:plugin> .....	64
<b>Chapter 3: JSP Pages as XML Documents .....</b>	<b>67</b>
Why an XML Representation .....	67
Transforming a JSP Page into an XML Document.....	68
List of Alternative XML Elements .....	68
Additional Description .....	69
DTD for the XML document .....	69
The jsp:root Element .....	69

Quoting Conventions .....	69
Request-Time Attribute Expressions .....	70
<b>Chapter 4: The JSP Engine .....</b>	<b>71</b>
The JSP Model .....	71
JSP Page Implementation Class.....	73
API Contracts .....	74
Request and Response Parameters .....	74
Omitting the extends Attribute .....	75
Using the extends Attribute.....	78
Buffering.....	78
<b>Chapter 5: Scripting Elements Based on Java .....</b>	<b>81</b>
Overall Structure.....	81
Declarations Section.....	83
Initialization Section .....	83
Main Section .....	83
<b>Appendix A: JSP Classes .....</b>	<b>85</b>
Package Description .....	85
The JSP Author Contract.....	85
The JspPage Interface.....	85
The HttpJspPage Interface .....	87
JspWriter .....	87
PageContext.....	94
JspFactory.....	102
<b>Appendix B: Java™ Servlet 2.1 clarification(s) .....</b>	<b>105</b>
Relative URL interpretation.....	105
Sharing Session State.....	106
Access Control.....	106

Path mapping .....	106
<b>Appendix C: Change History .....</b>	<b>109</b>
Changes Between 0.92 and 1.0 .....	109
Normalized Usage .....	109
Changes .....	109
Removals .....	110
Postponed for Evaluation in Future Releases .....	110
Additions .....	110
Changes between 0.91 and 0.92 .....	110
<b>Appendix D: Future Directions .....</b>	<b>111</b>
JSP 1.1 .....	111
Optional Features Become Mandatory .....	111
Tag Extension Mechanism .....	111
Additional Features .....	112
Support for J2EE 1.0 .....	112



# Preface

---

This document, the *JavaServer Pages™ 1.0 Specification*, describes the page formats and the APIs available in the version 1.0 of the JavaServer Pages Standard Extension.

This is a draft version; page xi provides more details on the status of this document.

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

Please send any comments you may have to [jsp-comments@calterra.eng.sun.com](mailto:jsp-comments@calterra.eng.sun.com)

## Who should read this document

This document is intended for:

- Web Server and Application Server vendors that want to provide JSP Engines that conform to the JSP 1.0 specification.
- Web Authoring Tool vendors that want to generate JSP pages that conform to the JSP 1.0 specification.
- Sophisticated JSP page authors that want to use advanced features like the `extends` directive.
- Eager JSP page authors who do not want to or cannot wait for Web Authoring Tools, or even for a User's Guide.

This document is not a User's Guide.

## Related Documents

The JavaServer Pages Specification relies on the framework provided by the Java™ Servlet specification. JSP 1.0 is based on small clarifications to the Java Servlet 2.1 specification.

JSP 1.0 requires only JDK 1.1 but it can take advantage of the Java 2 platform.

Implementors and authors of JSP will be interested in a number of other documents, of which the following are worth mentioning explicitly. Links to the documents will be available at our web site (<http://java.sun.com/products/jsp>):

- White paper on JSP.
- Java Servlet 2.1 Specification.
- HTML
- XML
- JavaBeans™ specification
- Java™ 2 Enterprise Edition documents

## Future Directions

JSP 1.1 is the follow-up release to JSP 1.0. JSP 1.1 is part of the Java 2 Enterprise Edition (J2EE) platform and will be accompanied by an update to the Java Servlet specification.

More details on future directions for JavaServer Pages are described in Appendix D.

---

# Font Conventions

TABLE P-1 Meaning of Typefaces Used in This Document

Typeface or Symbol	Meaning	Example
AaBbCc123	The names of commands, files, and directories; on-screen computer output	Edit your <code>.login</code> file. Use <code>ls -a</code> to list all files. <code>machine_name% You have mail.</code>
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type <code>rm filename</code> .
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in <i>User's Guide</i> . These are called <i>class</i> options. You <i>must</i> be root to do this.

---

# Acknowledgments

Many people contributed to the JavaServer Pages specification and reference implementation during its gestation period.

We want to thank the following people from Sun Microsystems: Anselm Baird-Smith, Dave Brownell, David-John Burrowes, Abhishek Chauhan, James Davidson, Satish Dharmaraj, Mala Chandra, Graham Hamilton, Mark Hapner, Vlada Matena, Mandar Raje, Bill Shannon, Joy Schiffner, James Todd, Vanitha Venkatraman, Anil Vijendran, and Connie Weiss.

The success of the Java Platform depends on the process used to define and evolve it. This open process permits the development of high quality specifications in internet time and involves many individuals and corporations. Although it is impossible to list all the individuals who have contributed, we would like to give thanks explicitly to the following individuals: Elias Bayeh, Hans Bergsten, Dave Brown, Bjorn Carlston, Shane Claussen, Mike Conner, Scott Ferguson, Bob Foster, Mike Freedman, Chris Gerken, Don Hsi, Jason Hunter, Amit Kishnani, Sanjeev Kumar, Rod McChesney, Roberto Mameli, Adam Messinger, Vincent Partington, Tom Rilley, Brian Surkan, Alex Yiu and Tom Yonkman. Apologies to any we may have missed.

Last, but certainly not least important, we thank the software developers and members of the general public who have read this specification, used the reference implementation, and shared their experience. You are the reason JavaServer Pages exists.



# Overview

---

---

## 1.1 Why JavaServer Pages?

JavaServer Pages™ is the Java™ platform technology for building applications containing dynamic Web content such as HTML, DHTML, XHTML and XML. JavaServer Pages allows you to write Web pages that create dynamic content as easily as possible but with maximum power and flexibility.

JavaServer Pages offers the following advantages:

- *Separation of dynamic and static content*

JavaServer Pages enable the separation of static content from dynamic content that is inserted into the static template. This greatly simplifies the creation of the content.

- *Support for scripting and tags*

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

- *Write Once, Run Anywhere™*

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

- *High quality tool support*

The Write Once, Run Anywhere properties of JSP allows the user to choose *best-of-breed* tools. Additionally, an explicit goal of the JavaServer Pages design is to enable the creation of high quality portable tools. JSP 1.0 provides the foundation which is enhanced in JSP 1.1 with the tag extension mechanisms and deployment and installation support.

- *Reuse of components and tags*

JavaServer Pages emphasizes the use of reusable components such as; JavaBeans, Enterprise JavaBeans and custom tags. These components can be used in interactive tools for component development and page composition. This saves you considerable development time while giving you the cross-platform power and flexibility of Java and other scripting languages.

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

JavaServer Pages is an integral part of the Java™ 2 Enterprise Edition (J2EE), which brings Java technology to enterprise computing. You can now develop powerful middle-tier server applications, using a JavaServer Pages Web site as a front end to Enterprise JavaBeans™ running in a J2EE compliant environment.

---

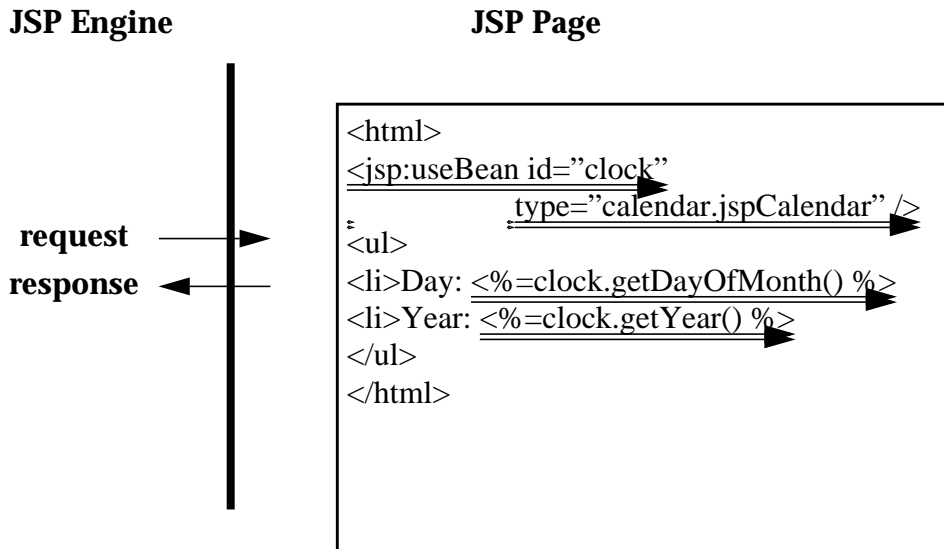
## 1.2 What is a JSP Page?

A JSP page is a text-based document that describes how to process a *request* to create a *response*. The description intermixes template data with some dynamic actions and leverages on the Java Platform.

### *An Example*

A simple example of a JSP page is shown in FIGURE 1-1. The example shows the response page, which is intended to be a short list with the day of the month and year at the moment when the request is received. The page itself contains several *fixed template text*, and some JSP elements that are shown underlined in the figure. The first element creates a Bean named `clock` of type `calendar.jspCalendar` and the next two use the Bean to display some of its properties.

FIGURE 1-1 An Example of a JSP Page.



### *The Servlet and JSP Standard Extensions*

JavaServer Pages is a Standard Extension that is defined on top of the Servlet Standard Extension. JSP 1.0 uses the classes from Java Servlet 2.1 specification. The changes are described in Appendix B, "Java™ Servlet 2.1 clarification(s)". JSP 1.0 and Servlet 2.1 rely only on features in the Java Runtime Environment 1.1, although they are compatible with the Java 2 Runtime Environment.

---

## 1.3 Features in JSP 1.0

The JSP 1.0 specification has mandatory and optional features. JSP 1.0 enables a tag extension mechanism for the creation of custom tags but such a mechanism will not appear until the JSP 1.1 specification.

The JSP 1.0 specification includes:

- JSP standard directives

- JSP standard actions
- Script language declarations, scriptlets and expressions

There is only one optional feature in the JSP 1.0 specification:

- Run-time attribute values.

JSP 1.1 will make the optional features of JSP 1.0 mandatory and it will add additional features to enhance their utility in a J2EE platform.

---

## 1.4 The JSP Model

A JSP page is executed by a JSP engine, which is installed on a Web server, or on a Web enabled application server. The JSP engine delivers *requests* from a client to a JSP page and *responses* from the JSP page to the client. The semantic model underlying JSP pages is that of a servlet: a JSP page describes how to create a response object from a request object for a given protocol, possibly creating and/or using in the process some other objects.

All JSP engines must support HTTP as a protocol for requests and responses, but an engine may also support additional request/response protocols. The default request and response objects are of type `HttpServletRequest` and `HttpServletResponse`, respectively.

A JSP page may also indicate how some events are to be handled. In JSP 1.0 only *init* and *destroy* events can be described: the first time a request is delivered to a JSP page a *jspInit()* method, if present, will be called to prepare the page. Similarly, a JSP engine can reclaim the resources used by a JSP page at any time that a request is not being serviced by the JSP page by invoking first its *jspDestroy()* method; this is the same life-cycle as that of *Servlets*.

JSP pages are often implemented using a JSP *translation phase* that is done only once, followed by some *request processing phase* that is done once per request. The translation phase usually creates a class that implements the `javax.servlet.Servlet` interface. The translation of a JSP source page into a corresponding Java implementation class file by a JSP engine can occur at any time between initial deployment of the JSP page into the runtime environment of a JSP engine, and the receipt and processing of a client request for the target JSP page.

A JSP page contains some *declarations*, some *fixed template* data, some (perhaps nested) *action instances*, and some *scripting elements*. When a request is delivered to a JSP page, all these pieces are used to create a response object that is then returned to the client. Usually, the most important part of this response object is the result stream.

## 1.4.1 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 Chapter 2, “Implicit Objects”); other objects are created explicitly through actions; objects can also be created directly using scripting code, although this is less common. The created objects have a *scope attribute* defining *where* there is a reference to the object and *when* that reference is removed.

The created objects may also be visible directly to the scripting elements through some scripting-level variables (see Section 1.4.5, “Objects and Variables”).

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

Objects are always created within some JSP page instance that is responding to some *request* object. JSP defines 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 (see Chapter 2, “Implicit Objects”).
- *request* - Objects with *request* scope are accessible from pages processing the same request where they were created. All references to the object shall be released after the request is processed; in particular, if the request is forwarded to a resource in the same runtime, the object is still reachable. References to objects with *request* scope are stored in the *request* object.
- *session* - Objects with *session* scope are accessible from pages processing requests that are in the same session as the one in which they were created. It is not legal to define an object with session scope from within a page that is not session-aware (see Section 2.8.1, “The page Directive”). All references to the object shall be released after the associated session ends. References to objects with *session* scope are stored in the *session* object associated with the page activation.
- *application* - Objects with *application* scope are accessible from pages processing requests that are in the same application as they one in which they were created. All references to the object shall be released when the runtime environment reclaims the `ServletContext`. Objects with application scope can be defined (and reached) from pages that are not session-aware (see Section 2.8.1, “The page Directive”). References to objects with *application* scope are stored in the *application* object associated with a page activation.

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

## 1.4.2 Fixed Template Data

*Fixed template data* is used to describe those pieces that are to be used *verbatim* either in the response or as input to JSP actions. For example, if the JSP page is creating a presentation in HTML of a list of, say, books that match some search conditions, the template data may include things like the `<ul>`, `</ul>`, and something like `<li>The following book...`

This fixed template data is written (in lexical order) unchanged onto the output stream (referenced by the implicit *out* variable) of the response to the requesting client.

## 1.4.3 Directives and Actions

*JSP elements* can be *directives* or *actions*. *Directives* provide global information that is conceptually valid independent of any specific request received by the JSP page. For example, a directive can be used to indicate the scripting language to use in a JSP page. *Actions* may, and often will, depend on the details of the specific request received by the JSP page. If a JSP is implemented using a compiler or translator, the directives can be seen as providing information for the compilation/translation phase, while actions are information for the subsequent request processing phase.

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

Directive elements have a syntax of the form

```
<%@ directive ...%>
```

There is also an alternative syntax that follows the XML syntax.

Action elements follow the syntax of XML elements, i.e. have a start tag, a body and an end tag:

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

or an empty tag

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

A JSP element has an *element type* describing its tag name, its valid attributes and its semantics; we refer to the type by its tag name.

## *Tag Extension Mechanism*

JSP 1.1 has a Tag Extension mechanism that enables the addition of new directives and actions, thus allowing the JSP “language” to be easily extended in a *portable* fashion. A typical example would be elements to support embedded database queries.

A tag library defines JSP element types and Customizers that allow these elements to be exposed as design time controls in page composition tools. Tag libraries can be used by JSP authoring tools and can be distributed along with JSP pages to any JSP runtime like Web and Application servers.

The Tag Extension mechanism assumes a Java RunTime environment.

Custom actions and directives defined using the Tag Extension mechanism follow the same semantic model described in this chapter.

## 1.4.4 Scripting Languages

*Scripting elements* are commonly used to manipulate objects and to perform computation that effects the content generated. There are three classes of scripting elements: *declarations*, *scriptlets* and *expressions*. *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 then inserted into the output stream.

All JSP engines must support scripting elements written in Java. Additionally, JSP engines 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 exceptions.

The precise definition of the semantics for Java scripting is given in Chapter 5.

The semantics for non-Java 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.

## 1.4.5 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 in two places: after its start tag and after its end tag. The variables will contain at process request-time a reference to the object defined by the element, although other references exist depending on the *scope* of the object (see Section 1.4.1, “Objects and Scopes”).

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, Java exposes properties via *getter* and *setter* methods, while these are available directly in JavaScript

The exact rules for the visibility of the variables are scripting language specific. Chapter 5 defines the rules for when the `language` attribute of the `jsp` directive is “java”.

## 1.4.6 JSP, HTML, and XML

The JSP specification is designed to support the dynamic creation of several types of structured documents, especially those using HTML and XML.

In general, a JSP page uses some data sent to the server in an HTTP request (for example, by a `QUERY` argument or a `POST` method) to interact with information already stored on the server, and then dynamically creates some content which is then sent back to the client. The content can be organized in some standard format (like HTML, DHTML, XHTML, XML, etc.), in some ad-hoc structured text format, or not at all.

XML is particularly useful with JSPs because of the extensibility and structure present in XML. See Section 1.6.4, “Using XML with JSP.”

There is another relationship between JSP and XML: a JSP page has a standard translation into a valid XML document. This translation is useful because it provides a standard mechanism to use XML tools and APIs to read, manipulate, and author JSP documents. The translation is defined in Chapter 3. JSP 1.0 processors are not required to accept JSP pages in this standard XML syntax, but this will be required in the JSP 1.1 specification.

---

## 1.5 A Web Application

A prototypical Web application can be composed from:

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

We expect that Web Servers supporting JSP 1.0 will support some notion like this, but JSP 1.0 has no specifications for portable packaging or deployment of such applications. We plan to add these notions for JSP 1.1.

### *URL Mappings*

A Web application is structured as a set of (possibly disjoint) mappings between these resources (above) and the URL namespace(s) of one, or more, HTTP servers.

For example, `http://www.myco.com/estore` may be the prefix for a map onto:

```
servlet/login.class
jsp/catalog.jsp
jsp/order.jsp
beans/shoppingcart.class
httpdocs/index.html
...
```

This mapping is implementation dependent in JSP 1.0; again, we anticipate that JSP 1.1 shall include a portable mechanism for specifying this mapping at application deployment time.

HTTP client(s) “invoke” applications by making HTTP requests (GET, POST) on these URLs, the server is responsible for mapping the requested URL to the appropriate resource and dispatching/handling the request and subsequent response as appropriate.

## *Applications and ServletContexts*

In JSP 1.0 (and Servlet 2.1) an HTTP protocol application is identified by a set of (possibly disjoint) URLs mapped to the resources therein. This URL set is associated, by the JSP engine (or Servlet runtime environment) with a unique instance of a `javax.servlet.ServletContext`. Servlets and JSPs in the same application can share this instance, and they can share global application state by sharing objects via the `ServletContext` `setAttribute()`, `getAttribute()` and `removeAttribute()` methods.

We assume that the information that a JSP page uses directly is all accessible from its corresponding `ServletContext`.

## *Sessions*

Each client (connection) may be assigned a session (`javax.servlet.HttpSession`) uniquely identifying it. Servlets and JSPs in the same “application” may share global session dependent state by sharing objects via the `HttpSession` `putValue()`, `getValue()` and `removeValue()` methods.

Care must be taken when sharing/manipulating such state between JSPs and/or Servlets since two or more threads of execution may be simultaneously active within Servlets and/or JSPs, thus proper synchronization of access to such shared state is required at all times to avoid unpredictable behaviors.

Note that sessions may be invalidated or expire at any time.

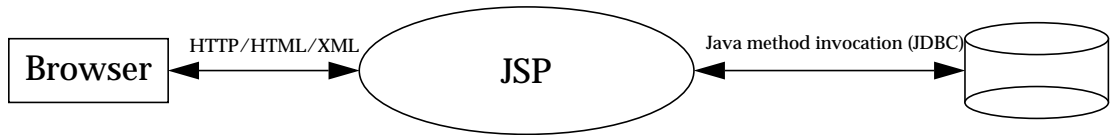
JSPs and Servlets handling the same `javax.servlet.ServletRequest` may pass shared state using the `ServletRequest` `setAttribute()`, `getAttribute()` and `removeAttribute()` methods.

---

# 1.6 Application Models

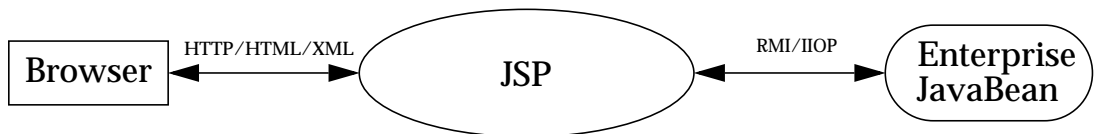
Java Server Pages can be used in combination with Servlets, HTTP, HTML, XML, Applets and Enterprise JavaBeans to implement a broad variety of application architecture(s) or models.

## 1.6.1 Simple 2<sup>1/2</sup>-Tier Application



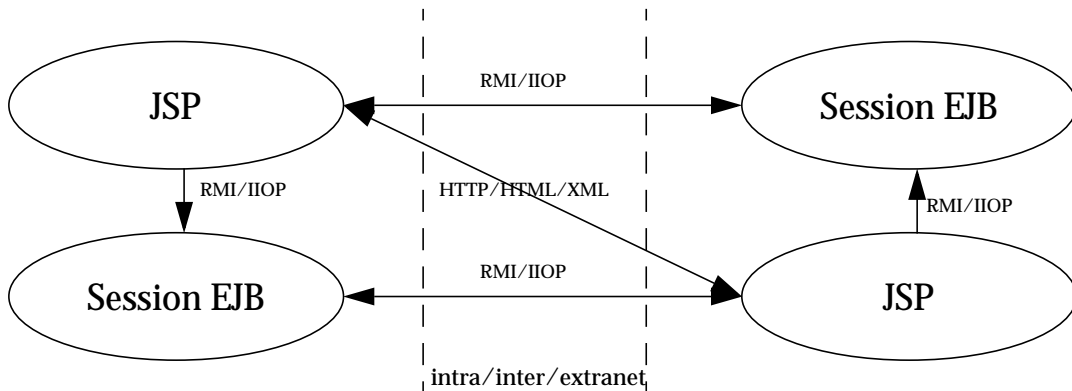
The simple 2-tier model (accessing a database in the example above) describes the cgi-bin replacement architecture that the Servlet model first enabled. This allows a JSP (or a Servlet) to directly access some external resource (such as a database or legacy application) to service a client's request. The advantage of such a scheme is that it is simple to program, and allows the page author to easily generate dynamic content based upon the request and state of the resource(s). However this architecture does not scale for a large number of simultaneous clients since each must establish/or share (ad-hoc) a (potentially scarce/expensive) connection to the resource(s) in question.

## 1.6.2 N-tier Application



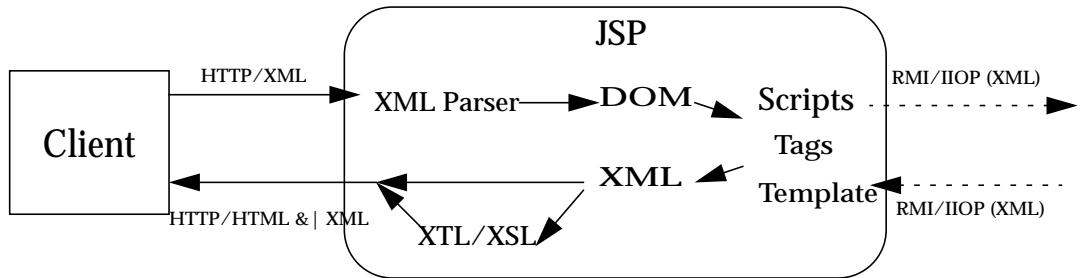
In this model the application is composed of ( $n \geq 3$ ) tiers, where the middle tier, the JSP, interacts with the back end resources via an Enterprise JavaBean. The Enterprise JavaBean server and the EJB provide managed access to resources thus addressing the performance issues. An EJB server will also support transactions and access to underlying security mechanisms to simplify programming. This is the programming model supported by the Java<sup>®</sup> 2 Enterprise Edition (J2EE) platform.

### 1.6.3 Loosely Coupled Applications



In this model we have two loosely coupled applications (either on the same Intranet, or over an Extranet or the Internet). These applications may be peers, or act as client or server for the other. A common example of this is supply chain applications between vendor enterprises. In such situations it is important that each participant be isolated from changes in the implementation of its dependents. In order to achieve this loose coupling the applications do not communicate using a fine grain imperative interface contract like those provided for by RMI/IIOP or JavaIDL. The applications communicate with each other via HTTP, using either HTML or XML to/from a JSP.

## 1.6.4 Using XML with JSP

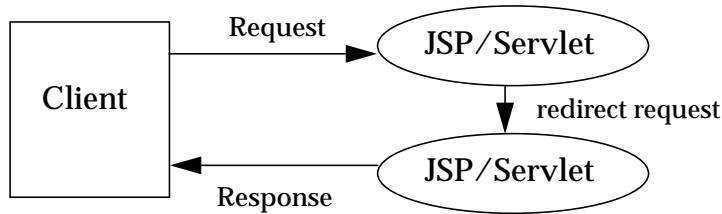


JavaServer Pages are an ideal way to describe processing of XML input and output. Simple XML generation can be done by just writing the XML as static template portions within the JSP. Dynamic generation will be done through JavaBean components, EJB components, or via custom tags that generate XML. Similarly, input XML can be received from POST or QUERY arguments and then sent directly to JavaBeans, EJB, or custom tags, or manipulated via the scripting.

There are two attributes of JSP that make it specially suited for describing XML processing. One is that XML fragments can be described directly in the JSP text either as templates for input into some XML-consuming component, or as templates for output to be extended with some other XML fragments. Another attribute is that the tag extension mechanism enables the creation of specific tags and directives that are targeted at useful XML manipulation operations.

JSP 1.1 will include several standard tags that will support XML manipulation, including the transformation of the XML produced by the given JSP using XTL/XSL.

## 1.6.5 Redirecting Requests

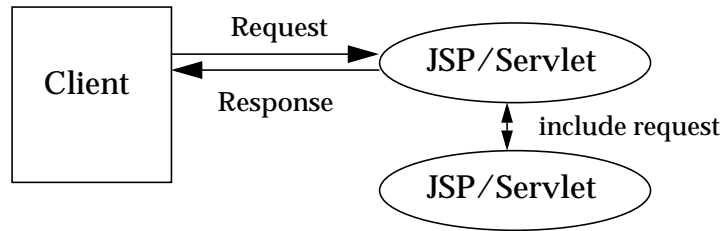


It is common that the data to be sent to the client varies significantly depending on properties of the client that are either directly encoded in the request object or can be discovered based on some user/client profile (e.g. stored in a login database). In this case it is very convenient to have the initial JSP determine details about the request and then, if necessary, redirect the request to a different JSP.

This programming model is supported by the underlying Servlet APIs. The properties of the HTTP protocol are such that the redirect cannot be done if the response stream has started being sent back to the client; this characteristic makes the description of some common situations quite inconvenient. To address this, JSP 1.0 by default provides buffering on the output stream. The JSP code can redirect the request at any point before flushing the output buffer.

Buffering is also very convenient for errorpage handling, since that is done by redirecting the request.

## 1.6.6 Including Requests



Another useful application model involves request includes. In this model, the request reaches an initial JSP page. The page may start generating/composing some result but at some point it may want to dynamically include the contents of some other page. These contents may be static but may also be dynamically generated by some other JSP, Servlet, or some legacy mechanism like ASP.

Although in some cases this inclusion model is applicable to presentation-dependent contents, it is most often used in the context of a presentation-independent content, like when the data generated is actually XML (which may be converted later into some other format using, say, XSL).



# Core Syntax and Semantics

---

This chapter describes the core syntax and semantics of the JavaServer Pages 1.0 Specification.

---

## 2.1 General Syntax Rules

The following general syntax rules apply to all elements in JSP pages.

### 2.1.1 JSP Elements and Template Data

A JSP page has some *JSP elements* and some *template data*. The JSP elements are instances of some *JSP element types* that are known to the JSP engine; *template data* is everything else: i.e. anything that the JSP engine does not understand.

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

There are three types of JSP elements: *directive elements*, *scripting elements*, and *action elements*; the corresponding syntax is described below. Template data is uninterpreted; it is usually passed through to the client, or to some processing component.

## 2.1.2 JSP Element Syntax

Most of the JSP syntax is based on XML. Elements based on the XML syntax have either a start tag (including the element name) possibly with attributes, an optional body, and a matching end tag, or they have an empty tag possibly with attributes:

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

and

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

Scripting elements and directives are written using a syntax that is easier to author by hand. Elements using the alternative syntax are of the form `<%.....%>`.

All JSP pages have an equivalent valid XML document. JSP 1.1-compliant engines will be required to accept JSP pages as well as their equivalent XML documents. Elsewhere this chapter describes the XML equivalent syntax for the scripting elements and directives; these XML element types are not intended to be used within a JSP page but in the equivalent XML document, as described in Chapter 3.

## 2.1.3 Start and End Tags

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

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

## 2.1.4 Empty Elements

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

## 2.1.5 Attribute Values

Following the XML specification, attribute values always appear quoted. Both single and double quotes can be used. The entities `&apos;` and `&quot;` are available to describe single and double quotes.

See also Section 2.13.1, "Request Time Attribute Values."

## 2.1.6 White Space

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

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

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

For this input,

	<code>&lt;?xml version="1.0" ?&gt;</code>
This is the default value	<code>&lt;%@ page buffer="8kb" %&gt;</code>
	The rest of the document goes here

The result is

	<code>&lt;?xml version="1.0" ?&gt;</code>
note the empty line	
	The rest of the document goes here

As another example, for this input,

	<code>&lt;% response.setContentType("...");</code>
note no white between the two elements	<code>whatever... %&gt;&lt;?xml version="1.0" ?&gt;</code>
	<code>&lt;%@ page buffer="8kb" %&gt;</code>
	The rest of the document goes here

The result is

no leading space	<code>&lt;?xml version="1.0" ?&gt;</code>
note the empty line	
	The rest of the document goes here

---

## 2.2 Error Handling

There are two logical phases in the lifecycle/processing of a JavaServer Page source file:

- Translation (or compilation) from JSP source into a Java Language class file.
- Per client request processing by an instance of the Java Language class.

Errors may occur at any point during processing of either phase. This section describes how such errors are treated by a compliant implementation.

### 2.2.1 Translation Time Processing Errors

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

### 2.2.2 Client Request Time Processing Errors

During the processing of client requests, arbitrary runtime errors can occur in either the body of the target JSP implementation or in some other code (Java or other implementation language) called from the body of the JSP. Such errors are realized in the page implementation using the Java Language exception mechanism to signal their occurrence to caller(s) of the offending behavior<sup>1</sup>.

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

However, any uncaught exceptions thrown from the body of the JSP implementation class result in the forwarding of the client request and uncaught exception to the `errorpage` URL specified by the offending JSP (or the implementation default behavior, if none is specified).

1. Note that this is independent of scripting language; this requires that unhandled errors occurring in a JSP's scripting language environment are to be signalled to a JSP's Java implementation class via the Java language exception mechanism.

The offending `java.lang.Throwable` describing the error that occurred is stored in the `javax.servlet.Request` instance for the client request using the `putAttribute()` method, using the name “`javax.servlet.jsp.JspException`”.

If the `<%@ jsp ... %>` directive `errorpage` attribute names a URL that refers to another JSP, and that JSP indicates that it is an error page (by setting the `jsp` directive’s `iserrorpage` attribute to `true`) then the “exception” implicit scripting language variable of that page is initialized to the offending `Throwable` reference.

## 2.2.3 Unimplemented Optional Feature

If an unimplemented optional feature is used, a fatal translation error will be issued.

The only optional feature in JSP 1.0 is described in Section 2.13.1, “Request Time Attribute Values.”

---

## 2.3 Comments

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

### *Generating Comments in Output to Client*

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

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

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

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

### *Commenting the JSP Page*

JSP 1.0 has no special mechanism to describe comments within a `.jsp` file, but the desired effect can be achieved by using a scriptlet. For example:

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

These comments are processed by the JSP engine and generate no data in the dynamically generated output stream response stream to a requesting client.

---

## 2.4 Quoting and Escape Conventions

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

### *Quoting in Scripting Elements*

- A literal %> is quoted by %\>

### *Quoting in Template Text*

- A literal <% is quoted by <\%

### *Quoting in Attributes*

- A ' is quoted as \'
- A " is quoted as \"
- A %> is quoted as %\>
- A <% is quoted as <\%

### *XML Representation*

The quoting conventions are different to those of XML. Chapter 3 describes the details of the transformation.

---

## 2.5 Web Applications

The JSP 1.0 specification provides a simple notion of a Web Application (Section 1.5) as a collection of resources, including JSP pages, Java Servlets, server-side Java classes, client-side Java Applets, JavaBeans and other classes, static pages, and other resources. The JSP 1.0 specification does not provide a portable mechanism for specifying this collection and their mapping; such a mechanism will be part of JSP

1.1. The JSP 1.0 specification requires that all these resources are implicitly associated with and accessible through a unique `ServletContext` instance, which is available as the `appContext` implicit object (Section 2.9).

Some useful clarifications on the semantics of the Java Servlet 2.1 specification are available in Appendix B.

In addition to explicit operations through the `appContext` object, the application to which a JSP page belongs is reflected in:

- The `include` directive (Section 2.8.3)
- The `jsp:include` action element (Section 2.14.4).
- The `include` and `dispatch` attributes in the `jsp:request` action (Section 2.14.5).

## 2.5.1 Relative URL Specifications within an Application

This specification contains several references to *relative URL specifications*. These specifications are as in RFC 2396 specification; i.e. only the path part, no scheme nor authority. Some examples are:

```
"myErrorPage.jsp"  
"/errorPages/SyntacticError.jsp"  
"/templates/CopyrightTemplate.html"
```

When such a specification starts with a `"/`, it is to be interpreted relative to the docroot of the Application in which the JSP page belongs; i.e. its `ServletContext` object provides the base context URL.

When such a specification does not start with a `"/`, it is to be interpreted relative to the current JSP page; its URL is the base context URL.

---

## 2.6 Overview of Semantics

A JSP page is executed by a JSP engine; the semantic model is that of a Servlet: a 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 (see Chapter 4, “The JSP Engine for details).

A JSP page implementation 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 5. Most details are not language specific and are described in this chapter.

Most of the content of a JSP page is devoted to describing what data is written into the output stream of the response (usually sent back to the client). The description is based on a `JspWriter` object that is exposed through the implicit object *out* (see Section 2.9, “Implicit Objects”). Its value varies:

- Initially, *out* is a new `JspWriter` object. This object may be different from the stream object from `response.getWriter()`, and may be considered to be interposed on the latter in order to implement buffering (see Section 2.8.1, “The page Directive”). This is the *initial out object*. JSP authors are prohibited from writing directly to either the `PrintWriter` or `OutputStream` associated with the `ServletResponse`.
- Within the body of some actions, *out* may be temporarily re-assigned to a different (nested) instance of `JspWriter` object. Whether this is or is not the case depends on the details of the actions semantics. Typically the content, or the results of processing the content, of these temporary streams is appended to the stream previously referred to by *out*, and *out* is subsequently re-assigned to refer to that previous (nesting) stream. Such nested streams are always buffered, and require explicit flushing to a nesting stream or discarding of their contents.
- If the *initial out* `JspWriter` object is buffered, then depending upon the value of the `autoflush` attribute of the `page` directive, the content of that buffer will either be automatically flushed out to the `ServletResponse` output stream to obviate overflow, an exception shall be thrown to signal buffer overflow. If the *initial out* `JspWriter` is unbuffered, then content written to it will be passed directly through to the `ServletResponse` output stream.

A JSP page can also describe what should happen when some specific *events* occur. In JSP 1.0, the only events that can be described are initialization and destruction of the page; these are described using “well-known method names” in declaration elements (see page 72). JSP 1.1 will likely define more events as well as a more structured mechanism for describing the actions to take.

---

## 2.7 Template Text Semantics

The semantics of *template (or uninterpreted) Text* is very simple: the template text is passed through to the current *out* `JspWriter` implicit object, after applying the substitutions of Section 2.4, “Quoting and Escape Conventions”.

---

## 2.8 Directives

Directives are messages to the JSP engine. In JSP 1.0, 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. The XML alternative syntax for directives is:

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

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

The remainder of this section describes the standard directives that are available on all conforming JSP 1.0 implementations.

## 2.8.1 The page Directive

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

A translation unit (JSP source file and any files included via the `include` directive) can contain more than one instance of the `jsp` directive, all the attributes will apply to the complete translation unit (i.e. page directives are position independent). However, there shall be only one occurrence of any attribute/value defined by this directive in a given translation unit with the exception of the “`import`” attribute; multiple uses of this attribute are cumulative (with ordered set union semantics). Other such multiple attribute/value (re)definitions result in a fatal translation error.

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

Unrecognized attributes or values result in fatal translation errors.

### *Examples*

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

```
<%@ page info="my latest JSP Example V1.0" %>
```

The following directive requests no buffering, indicates that the page is thread safe, and provides an error page.

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

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

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

The XML syntax for the first example is:

```
<jsp:directive.page info="my latest JSP Example V1.0" />
```

### 2.8.1.1 JSP Syntax

```
<%@ page page_directive_attr_list %>
```

```
page_directive_attr_list ::=      { language="scriptingLanguage" }  
                                { extends="className"           }  
                                { import="packagelist"          }  
                                { session="true|false"           }  
                                { buffer="none|sizekb"           }  
                                { autoflush="true|false"         }  
                                { isthreadsafe="true|false"      }  
                                { info="info_text"               }  
                                { errorpage="error_url"          }  
                                { iserrorpage="true|false"       }
```

The details of the attributes are as follows:

language	<p>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 <code>include</code> directive below).</p> <p>In JSP 1.0, the only defined and required scripting language value for this attribute is “java”. <u>This specification only describes the semantics of scripts for when the value of the language attribute is “java”.</u></p> <p>When “java” is the value of the scripting language, the Java Language source code fragments used within the translation unit are required to conform to the Java Language Specification; as indicated in Chapter 5, “Scripting Elements Based on Java” .</p> <p>All scripting languages must provide some implicit objects that a JSP developer can use in declarations, scriptlets, and expressions. The specific objects that can be used are defined in Section 2.9, “Implicit Objects.”</p> <p>All scripting languages must support the Java Runtime Environment (JRE). All scripting languages must expose the Java object model to the script environment, especially implicit variables, JavaBeans properties, and public methods.</p> <p>Future versions of the JSP specification may define additional values for the language attribute and all such values are reserved.</p> <p>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.</p>
extends	<p>The value is a fully qualified Java class name, that names the superclass of the Java class to which this JSP page is transformed (see Chapter 5).</p> <p>This attribute should not be used without careful consideration as it restricts the ability of the JSP Engine to provide specialized superclasses that may improve on the quality of rendered service.</p>

import	<p>The value is a (comma separated) list of one or more Java package names that describe packages that the translated JSP page implementation class shall import and are thus available to the scripting environment.</p> <p>This value is currently only defined when the value of the language directive is "java".</p>
session	<p>Indicates that the page requires participation in an (http) session.</p> <p>If "true" then the implicit script language variable named "session" of type <code>javax.servlet.HttpSession</code> references the current/new session for the page.</p> <p>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 is illegal and shall result in a fatal translation error.</p> <p>Default is "true".</p>
buffer	<p>Specifies the buffering model for the initial "out" <code>JspWriter</code> to handle content output from the page.</p> <p>If "none", then there is no buffering and all output is written directly through to the <code>ServletResponse PrintWriter</code>.</p> <p>If a buffer size is specified (e.g 12kb) then output is buffered with a buffer size not less than that specified.</p> <p>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.</p> <p>The default is buffered with an implementation buffer size of not less than 8kb.</p>
autoflush	<p>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.</p> <p>The default is "true".</p> <p>Note: it is illegal to set autoflush to "false" when "buffer=none".</p>

<code>isthreadsafe</code>	<p>Indicates the level of thread safety implemented in the page.</p> <p>If “<code>false</code>” then the JSP processor shall dispatch multiple outstanding client requests, one at a time, in the order they were received, to the page implementation for processing.</p> <p>If “<code>true</code>” then the JSP processor runtime may choose to dispatch multiple outstanding client requests to the page simultaneously.</p> <p>Page authors using “<code>true</code>” must ensure that they properly synchronize access to page shared state.</p> <p>Default is “<code>true</code>”.</p> <p>Note that even if the <i>isthreadsafe</i> attribute is “<code>false</code>” the JSP author must ensure that access to any shared objects shared in either the <code>ServletContext</code> or the <code>HttpSession</code> are properly synchronized.<sup>1</sup></p>
<code>info</code>	<p>Defines an arbitrary string that is incorporated into the translated page, that can subsequently be obtained from the page’s implementation of <code>Servlet.getServletInfo()</code> method.</p>
<code>iserrorpage</code>	<p>Indicates if the JSP is intended to be the URL target of another JSP’s <code>errorpage</code>.</p> <p>If “<code>true</code>”, then the implicit script language variable “<code>exception</code>” is defined and its value is a reference to the offending <code>Throwable</code> from the source JSP in error.</p> <p>If “<code>false</code>” then the “<code>exception</code>” implicit variable is unavailable, and any reference to it within the body of the JSP is illegal and shall result in a fatal translation error.</p> <p>Default is “<code>false</code>”</p>

`errorpage` Defines a URL to a resource to which any Java `Throwable` object(s) thrown but not caught by the page implementation are forwarded to for error processing.

The provided `URLSpec` is as in Section 2.5.1.

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

If the URL names another JSP then, when invoked that JSP's `exception` implicit script variable shall contain a reference to the originating uncaught `Throwable`.

The default URL is implementation dependent.

Note the `Throwable` object is transferred by the throwing page implementation to the error page implementation by saving the object reference on the common `ServletRequest` object using the `setAttribute()` method, with a name of `"javax.servlet.jsp:jspException"`.

Note: if `autoflush=true` then if the contents of the initial `JspWriter` has been flushed to the `ServletResponse` output stream then any subsequent attempt to dispatch an uncaught exception from the offending page to an `errorpage` may fail.

1. If `isthreadsafe="true"`, the JSP page implementation shall implement `javax.servlet.SingleThreadModel`. Some implementation(s) may additionally use a pool of page implementation instances to do load balancing therefore a page author cannot assume that all requests mapped to a particular JSP shall be delivered to the same instance of that page's implementation class.

## 2.8.1.2 XML Syntax

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

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

See above for description of `page_directive_attr_list`.

## 2.8.2 Including Data in JSP Pages.

Including data is a significant part of the tasks in a JSP page. Accordingly, the JSP 1.0 specification has several include mechanisms suited to different tasks. A summary of their semantics is shown in TABLE 2-1.

TABLE 2-1 Summary of Include Mechanisms in JSP 1.0

Syntax	Phase	Description	Section
<code>&lt;%@ include ... %&gt;</code>	translation-time	Content is parsed by JSP processor.	2.8.3
<code>&lt;jsp:include file= /&gt;</code>	request-time	Content is not parsed; it is included in place.	2.14.4
<code>&lt;jsp:request include= /&gt;</code>	request-time	Request is passed; result from that process is included in place.	2.14.5

## 2.8.3 The include Directive

The `include` directive is used to substitute text and/or code at JSP translation-time. The `<%@ include file="relativeURLspec" %>` directive inserts the text of the specified resource into the `.jsp` file. The included file is subject to the access control available to the JSP engine. The `file` attribute is as in Section 2.5.1.

A JSP engine can include a mechanism for being notified if an included file changes, so the engine can recompile the JSP page. However, JSP 1.0 does not have a way of directing the JSP engine that included files have changed.

### *Examples*

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

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

### 2.8.3.1 JSP Syntax

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

### 2.8.3.2 XML Syntax

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

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

## 2.8.4 The taglib Directive

The set of significant tags a JSP engine interprets can be extended to include custom tags with their own semantics. In this version of the specification the implementation mechanism to enable this is not defined. A portable custom tag extension definition and customization mechanism will be defined for a future version of this specification.

This directive declares that the page uses custom tags, uniquely names the (implementation dependent) "tag library" that they are defined in, and associates a tag prefix that will distinguish usage of those tags therein.

It is a fatal translation error for taglib directive to appear after actions using the prefix introduced by the taglib directive.<sup>1</sup>

### *Examples*

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

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

If a particular JSP engine implementation does not recognize the tag library named by the URI this will result in a fatal translation error.

1. This enables a simple 1-pass implementation for isolating template text from jsp actions and directives.

## 2.8.4.1 JSP Syntax

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

*tagLibraryURI*

Uninterpreted in this version of the specification; it uniquely names the set of custom tags associate with this prefix.

When a portable tag extension mechanism is defined in a future version of this specification, this URI shall be used to locate a portable implementation of the tag library implementing the tags used therein.

*tagPrefix*

Defines the *prefix* string in; `<prefix>:<tagname>` that is used to distinguish a custom tag instance, e.g:

```
<jsp:an_example_tag ... />
```

Tag prefixes `jsp:`, `jspx:`, `java:`, `javax:`, `servlet:`, `sun:`, and `sunw:` are reserved.

Empty prefixes are illegal in this version of the specification.

## 2.8.4.2 XML Syntax

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

```
<jsp:directive.taglib uri="tagLibraryURI" prefix="tagPrefix" />
```

---

# 2.9 Implicit Objects

When you author JSPs, you have access to certain implicit Java objects that are always available for use within scriptlets and expressions, without being declared first. All scripting languages are required to provide access to these objects.

Each implicit object has a class or interface type defined in a core Java or Java Servlet API package, as shown in TABLE 2-2.

**TABLE 2-2** Implicit Objects Available in JSP Pages

Implicit Variable	Of Type	What It Represents	Scope
request	protocol dependent subtype of: javax.servlet.ServletException e.g: javax.servlet.HttpServletRequest	The request triggering the service invocation.	request
response	protocol dependent subtype of: javax.servlet.ServletResponse e.g: javax.servlet.HttpServletResponse	The response to the request.	page
pageContext	javax.servlet.jsp.PageContext	the page context for this JSP	page
session	javax.servlet.http.HttpSession	The session object created for the requesting client (if any).	session
application	javax.servlet.ServletContext	This variable is only valid for Http protocols. The servlet context obtained from the servlet configuration object (as in the call <code>getServletConfig().getContext()</code> )	application
out	javax.servlet.jsp.JspWriter	An object that writes into the output stream.	page
config	javax.servlet.ServletConfig	The ServletConfig for this JSP	page
page	java.lang.Object	the instance of this page's implementation class processing the current request <sup>1</sup>	page

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

In addition, in an error page, you can access the exception implicit object, described in TABLE 2-3.

TABLE 2-3 Implicit Objects Available in Error Pages

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

## 2.10 The pageContext Object

The `pageContext` implicit object provides convenient access to the different name scopes available to the JSP page.

More examples of their uses will be available in the next draft of this specification; in the meantime refer to Appendix A or to the javadoc documentation.

## 2.11 Scripting Elements

JSP 1.0 has three scripting language elements—declarations, scriptlets, and expressions. A scripting language precisely defines the semantics for these elements but, informally, a declaration declares elements, a scriptlet is a statement fragment, and an expression is a complete language expression. The scripting language used in the current page is given by the value of the `language` directive (see Section 2.8.1, “The page Directive”). In JSP 1.0, the only value defined is “java”.

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 “%>”.

## 2.11.1 Declarations

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

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

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

### *Examples*

For example, the first declaration below, in JSP syntax, declares an integer, and initializes it to zero; while the second declaration, in XML syntax, declares a method. Note that the second example uses a CDATA statement to avoid having to quote the "<" inside the `jsp:decl`.

```
<%! int i = 0; %>
<jsp:decl> <![CDATA[ public void f(int i) { if (i<3)
out.println("..."); } ]]> </jsp:decl>
```

### 2.11.1.1 JSP Syntax

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

### 2.11.1.2 XML Syntax

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

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

Using a DTD, the corresponding declaration is:

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

## 2.11.2 Scriptlets

Scriptlets can contain any code fragments that are valid for the scripting language specified in the `language` directive. Whether the code fragment is legal depends on the details of the scripting language; see Chapter 5.

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

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

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

### *Examples*

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

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

#### 2.11.2.1 JSP Syntax

```
<% scriptlet %>
```

#### 2.11.2.2 XML Syntax

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

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

Using a DTD, the corresponding declaration is:

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

### 2.11.3 Expressions

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

If the result of the expression cannot be coerced to a `String` then a `ClassCastException` will be thrown.

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

The contents of a JSP expressions must be a complete expression in the scripting language in which they are written.

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

### *Examples*

In the next example, the current date is inserted.

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

#### 2.11.3.1 JSP Syntax

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

#### 2.11.3.2 XML Syntax

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

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

Using a DTD, the corresponding declaration is:

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

---

## 2.12 Actions

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

The JSP specification includes some action types that are *standard* and must be implemented by all conforming JSP engines. New action types are introduced using the `jsp:taglib` directive. JSP 1.1 will contain a tag extension mechanism to be used to describe the semantics of all non-standard actions<sup>1</sup>.

---

## 2.13 Tag Attribute Interpretation Semantics

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

### 2.13.1 Request Time Attribute Values

Action elements (both standard and custom) can define named attributes and associated values. Typically a JSP treats these values as fixed and immutable but the JSP 1.0 provides an optional mechanism to describe a value that is computed at request time.

An attribute value of the form "`<%= scriptlet_expr %>`" or '`<%= scriptlet_expr %>`' denotes a value that is computed at request time. The value denoted is that of the scriptlet expression involved. If there are more than one such attribute in a tag, the expressions are evaluated left-to-right.

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

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

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

This facility is optional in JSP 1.0 and mandatory in JSP1.1. Most attributes in the actions defined in JSP 1.0 have page translation-time semantics, the exceptions are:

- The `value` attribute of `jsp:setProperty` (2.14.2).
- The `file` attribute of `jsp:include` (2.14.4).
- The `dispatch` and `include` attributes of `jsp:request` (2.14.5).

1. The tag mechanism may also provide for custom directives.

## 2.13.2 The `id` Attribute

The `id="name"` attribute/value tuple in an element has special meaning to a JSP engine, both at page translation time, and at client request processing time; in particular:

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

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

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

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

Chapter 5 provides details for the case where the language attribute is "java".

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

For example:

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

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

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

### 2.13.3 The scope Attribute

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

<code>page</code>	<p>The named object is available from the <code>javax.servlet.jsp.PageContext</code> for the current page.</p> <p>This reference shall be discarded upon completion of the current request by the page body.</p> <p>It is illegal to change the instance object associated, such that its runtime type is a subset of the type of the current object previously associated.</p>
<code>request</code>	<p>The named object is available from the current page's <code>ServletRequest</code> object using: <code>ServletRequest.getAttribute(name)</code> method.</p> <p>This reference shall be discarded upon completion of the current client request.</p> <p>It is illegal to change the value of an instance object so associated, such that its runtime type is a subset of the type(s) of the object previously so associated.</p>

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

---

## 2.14 Standard Actions

JSP 1.0 defines some standard action types that are always available, regardless of the version of the JSP engine or Web server the developer uses. The standard action types are in addition to any custom types specific to a given JSP implementation (note that any such types, even if they are implemented non-portably in JSP 1.0, must be introduced via a `taglib` directive).

### 2.14.1 `<jsp:useBean>`

A `<jsp:useBean>` action makes an instance of a given `JavaBean`<sup>1</sup> class with a given `scope` available with a given `id`. The instance is either created or retrieved.

1. the class specified shall implement a public no-args constructor.

A usebean action performs four functions:

1. attempts to locate an object based on the attribute values (`id`, `class`, `scope`). The inspection is done synchronized.
2. defines a scripting language variable with the given `id` in the current lexical scope of the scripting language
3. if the object is found, initializes that variable's value with the located object.

If the `jsp:usebean` element had a non-empty body it is ignored.

4. if the object is not found, it creates the object (using the `class`), and then associates the object and name using the appropriate mechanism as specified by the `scope` attribute (see `PageContext`).

Afterwards, if the `jsp:useBean` element had a non-empty body, the body is processed. The variable is visible within the body. The text of the body is treated as elsewhere; if there is template text it will be passed through to the out stream; scriptlets will be evaluated, and action tags will too.

A common use of a non-empty body is to complete initializing the created bean; in that case the body will likely have `jsp:setProperty` actions and scriptlets.

It is a fatal translation error if these `jsp:setProperty` tag(s) specify an `id` attribute.

## *Examples*

In the following example, a bean with name "connection" of type "com.myco.myapp.Connection" is available after this element; either because it was already created or because it is newly created.

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

### 2.14.1.1 Syntax

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

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

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

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

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

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

<i>id</i>	The name of the Bean instance. The name you supply is case sensitive and must conform to the current scripting language variable-naming conventions.
<i>scope</i>	The scope within which the reference is available. The default value is <code>page</code> . See the description of the <code>scope</code> attribute defined earlier herein
<i>class</i>	The fully qualified name of the class that defines the Bean. The class name is case sensitive. This tag has translation time semantics, therefore it's value shall not be computed from a scriptlet expression

## 2.14.2 `<jsp:setProperty>`

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

There are two variants of the `jsp:setProperty` action. The first variant sets one or more properties in a Bean from one or more parameters in the request object. In this case, the property must either be a simple property of type `java.lang.String` or an indexed property of type `java.lang.String`, with the parameter matching these.

The second variant sets one property in a Bean from a constant or computed run-time expression. If a computed run-time expression is used, the types of the property and of the value must match, if a simple property; if the property is indexed, the type and length of the property must match that of the value. If a constant is used, then a conversion is attempted as shown in TABLE 2-4. A conversion failure leads to an error; the error may be at translation or at request-time.

**TABLE 2-4** Valid assignments in `jsp:setProperty`

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

## Examples

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

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

The following element sets a property from a value

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

### 2.14.2.1 Syntax

```
<jsp:setProperty name="beanName" prop_expr />
prop_expr ::=          property="*" |
                      property="propertyName" param="parameterName" |
                      property="propertyName" value="propertyValue"
```

*propertyValue* ::= *string*

Optionally, *propertyValue* can also be a request-time attribute value, as described in Section 2.13.1, "Request Time Attribute Values. This is optional in JSP 1.0 and will be mandatory in JSP 1.1.

*propertyValue* ::= *expr\_scriptlet*<sup>1</sup>

1. See syntax for expression scriptlet "<%= ... %>"

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

<i>name</i>	The name of a Bean instance defined by a <code>&lt;jsp:useBean&gt;</code> element or some other element. The Bean instance must contain the property you want to set. The defining element (in JSP 1.0 only a <code>&lt;jsp:useBean&gt;</code> element) must appear before the <code>&lt;jsp:setProperty&gt;</code> element in the same file.
<i>property</i>	<p>The name of the Bean property whose value you want to set</p> <p>If you set <i>propertyName</i> to * then the tag will iterate over the current <code>ServletRequest</code> 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.</p>
<i>param</i>	<p>The name of the request parameter whose value you want to give to a Bean property. The name of the request parameter usually comes from a Web form</p> <p>If you omit <i>param</i>, the request parameter name is assumed to be the same as the Bean property name</p> <p>If the <i>param</i> is not set in the Request object, or if it has the value of "", the <code>jsp:setProperty</code> element has no effect (a noop).</p> <p>An action may not have both <i>param</i> and <i>value</i> attributes.</p>
<i>value</i>	<p>The value to assign to the given property. It may be a page translation-time constant, or a request-time expression, when this feature is implemented.</p> <p>An action may not have both <i>param</i> and <i>value</i> attributes.</p>

### 2.14.3 `<jsp:getProperty>`

The `<jsp:getProperty>` tag places the value of a Bean instance property, converted to a `String`, into the implicit `out` object, from which you can display the value as output. The Bean instance must be defined as indicated in the *name* attribute before this point. Note that `<jsp:getProperty>` does not provide a *scope* attribute (see Section 1.4.1, "Objects and Scopes," on page 1-19).

## Examples

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

### 2.14.3.1 Syntax

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

The attributes are:

<i>name</i>	The name of the object instance from which the property is obtained.
<i>property</i>	Names the property to get.

### 2.14.4 <jsp:include>

A `<jsp:include .../>` element provides for the inclusion of static resources in the same context as the current page. See TABLE 2-1 for a summary of include facilities.

## Examples

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

### 2.14.4.1 Syntax

```
<jsp:include file="urlSpec" />
```

The attributes are:

<i>file</i>	The URL is a relative <i>urlSpec</i> as in Section 2.5.1.  If the page output is buffered then the buffer is flushed prior to forwarding.  Accepts a request-time attribute value (which must evaluate to a URL string).
-------------	--

## 2.14.5 <jsp:request>

A `<jsp:request .../>` element provides access to active resources in the same context as the current page. There are two variants of this element type, one allows inclusion, the other forwarding.

A `<jsp:request include="urlSpec" />` element allows the runtime inclusion of the result of applying JSP pages and Servlets to the current request. See TABLE 2-1 for a summary of include facilities.

A `<jsp:request forward="urlSpec" />` element allows the runtime dispatch of the current request to a JSP page or Java Servlet. A dispatch effectively terminates the execution of the current page.

In both variants, a relative *urlSpec* is as in Section 2.5.1.

### *Examples*

The following element might be used within a number of different pages to present a portfolio. This might be useful both for JSP pages producing HTML (say within a table, a section of a document, or some other well-contained area) as well as in the case of JSP pages producing XML that will later be processed before being sent to the client. In this case “portfolio.jsp” is a JSP page that is a sibling of the current JSP page

```
<jsp:request include="portfolio.jsp" />
```

### 2.14.5.1 Syntax

```
<jsp:request forward|include="relativeURLspec" />
```

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

---

include	<p>The current request(<code>ServletRequest</code>) and response (<code>ServletResponse</code>) is passed to the specified URL resource for processing. The resource specified can be a JSP, <code>Servlet</code>, or some other resource for which the runtime environment can provide a <code>RequestDispatcher</code> to.</p> <p>The resource may only write content to the <code>ServletResponse</code>.</p> <p>Request processing resumes in the calling JSP, once the “include” is completed.</p> <p>The URL is a relative <i>urlSpec</i> is as in Section 2.5.1.</p> <p>If the page output is buffered then the buffer is flushed prior to forwarding.</p> <p>Accepts a request-time attribute value (which must evaluate to a URL string).</p>
forward	<p>The current request (<code>ServletRequest</code>) and response (<code>ServletResponse</code>) is forwarded to the specified URL resource for processing. This terminates the request processing by the current page.</p> <p>If the page output is buffered then the buffer is flushed prior to forwarding.</p> <p>The URL is a relative <i>urlSpec</i> is as in Section 2.5.1.</p> <p>If the page output was unbuffered and anything has been written to it, an attempt to forward the request will result in an <code>IllegalStateException</code>.</p> <p>Accepts a request-time attribute value (which must evaluate to a URL string).</p>

---

## 2.14.6 <jsp:plugin>

The plugin action enables a JSP 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 (if required) and subsequent execution of the Applet or JavaBean 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. A valid implementation of this element in JSP 1.0 may send to the out stream the contents of the fallback body, or empty, if there is no immediate fallback subelement of this element.

### Examples

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

### 2.14.6.1 Syntax

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

<code>type</code>	Identifies the type of the component; a bean, or an applet.
<code>code</code>	As defined by HTML spec
<code>codebase</code>	As defined by HTML spec
<code>align</code>	As defined by HTML spec
<code>archive</code>	As defined by HTML spec
<code>height</code>	As defined by HTML spec
<code>hspace</code>	As defined by HTML spec
<code>jjreversion</code>	Identifies the spec version number of the JRE the component requires in order to operate; the default is: "1.1"
<code>name</code>	As defined by HTML spec
<code>vspace</code>	As defined by HTML spec
<code>title</code>	As defined by the HTML spec
<code>width</code>	As defined by HTML spec
<code>nspluginurl</code>	URL where JRE plugin can be downloaded for Netscape Navigator, default is implementation defined.
<code>iepluginurl</code>	URL where JRE plugin can be downloaded for IE, default is implementation defined.



## JSP Pages as XML Documents

---

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

---

### 3.1 Why an XML Representation

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

- An XML document must have a single top element; a JSP page is conveniently organized as a sequence of template text and JSP elements.
- In an XML document all tags are “significant”; to “pass through” a tag, it needs to be escaped using a mechanism like CDATA. In JSP tags that are unknown to the JSP processor are passed through automatically.
- Some very common programming tokens, like “<” are significant to XML; JSP provides a mechanism (the `<%` syntax) to “pass through” these tokens.

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

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

JSP 1.0 addresses both requirements by providing a friendly syntax and also defining a standard XML document for a JSP page. A JSP 1.0-compliant tool needs not do anything special with this document, but the JSP 1.1 specification is likely to require for JSP engines to accept both JSP pages and their equivalent XML documents.

---

## 3.2 Transforming a JSP Page into an XML Document

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

- Add a `<jsp:root>` element as the root. Enable a “jsp” namespace prefix for the standard tags within this root.
- Convert all the `<%` elements into valid XML elements as described in Chapter 2 (also see below).
- Convert the quotation mechanisms appropriately.
- Convert the `taglib` directive into namespace attributes of the `<jsp:root>` element.
- Create CDATA elements for all segments of the JSP page that do not correspond to JSP elements.

---

## 3.3 List of Alternative XML Elements

Chapter 2 described XML elements for the JSP elements that do not follow the XML syntax; they are summarized below:

**TABLE 3-1** XML standard tags for JSP directives and scripting elements

JSP element	XML equivalent
<code>&lt;%@ page ... %&gt;</code>	<code>&lt;jsp:directive.page ... /&gt;</code>
<code>&lt;%@ taglib ... %&gt;</code>	<code>&lt;jsp:directive.taglib ... /&gt;</code> , or jsp:root element is annotated with namespace information.
<code>&lt;%@ include ... %&gt;</code>	<code>&lt;jsp:directive.include ... /&gt;</code>
<code>&lt;%! ... %&gt;</code>	<code>&lt;jsp:decl&gt; .... &lt;/jsp:decl&gt;</code>
<code>&lt;% ... %&gt;</code>	<code>&lt;jsp:scriptlet&gt; .... &lt;/jsp:scriptlet&gt;</code>
<code>&lt;%= ... %&gt;</code>	<code>&lt;jsp:expression&gt; .... &lt;/jsp:expression&gt;</code>

---

## 3.4 Additional Description

---

**Note** – The XML mapping is still being defined. We encourage feedback and comments.

---

### 3.4.1 DTD for the XML document

A DTD for the standard XML document equivalent to a JSP page will be provided.

The proposed Document Type Declaration is:

```
<! DOCTYPE root
  PUBLIC "-//Sun Microsystems Inc.//DTD JavaServer Pages Version
  1.0//EN"
    "http://java.sun.com/products/jsp/dtd/jspcore_1_0.dtd">
```

### 3.4.2 The `jsp:root` Element

In addition to the XML elements described in Chapter 2 there is one additional element, `jsp:root`.

The `jsp:root` element is used as the top element of the standard XML document. It is also the place where taglibs will insert their namespace attributes. The top element has an `xmlns` attribute that enables the use of the standard JSP elements.

```
<jsp:root
  xmlns:jsp="http://java.sun.com/products/jsp/dtd/jsp_1_0.dtd">
  remainder of transformed JSP page
</jsp:root>
```

---

**Note** – The mapping for the `taglib` directive is not yet decided. Both an XML element and just adding additional `xmlns` attributes are being considered.

---

### 3.4.3 Quoting Conventions

The quoting rules for JSP 1.0 are designed to be friendly for hand authoring, they are not valid XML conventions.

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

### 3.4.4 Request-Time Attribute Expressions

The convention used in this specification for describing request-time attribute expressions produces attribute values that are not valid in XML documents. The conversion is not yet final at the time of writing this draft.

# The JSP Engine

---

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

This chapter is independent on the Scripting Language used in the JSP page. Chapter 5, “Scripting Elements Based on Java” provides the details specific to when the `language` directive has “java” as its value.

---

## 4.1 The JSP Model

As indicated in Section 1.4, “The JSP Model”, a JSP page is executed by a JSP engine, which is installed on a Web Server or Web enabled Application Server. The JSP engine delivers requests from a client to a JSP and responses from the JSP to the client. The semantic model underlying JSP pages is that of a Servlet: a JSP page describes how to create a *response* object from a *request* object for a given protocol, possibly creating and/or using in the process some other objects. A JSP page may also indicate how some events (in JSP 1.0 only *init* and *destroy* events) are to be handled.

### *The Protocol Seen by the Web Server*

The entity that processes *request* objects creating *response* objects should behave as if it were a Java class; in this specification we will simply assume it is a Java class. This Java class must implement the Servlet protocol. It is the role of the JSP engine to first locate the appropriate instance of such a class and then to deliver requests to it according to the Servlet protocol. As indicated elsewhere, a JSP engine may need to create such a class dynamically from the JSP page source before delivering a request and response objects to it.

Thus, Servlet defines the contract between the JSP engine and the Java class implementing the JSP page. When the HTTP protocol is used, the contract is described by the `HttpServlet` class. Most pages use the HTTP protocol, but other protocols are allowed by this specification.

### *The Protocol Seen by the JSP Page Author*

The JSP specification also defines the contract between the JSP engine and the JSP page author. This is, what assumptions can an author make for the actions described in the JSP page.

The main portion of this contract is the `_jspService()` method that is generated automatically by the JSP engine from the JSP page. The details of this contract is provided in Chapter 5.

The contract also describes how a JSP author can indicate that some actions must be taken when the `init()` and `destroy()` methods of the page implementation occur. In JSP 1.0 this is done by defining methods with name `jspInit()` and `jspDestroy()` in a declaration scripting element in the JSP page. Before the first time a request is delivered to a JSP page a `jspInit()` method, if present, will be called to prepare the page. Similarly, a JSP engine can reclaim the resources used by a JSP page at any time that a request is not being serviced by the JSP page by invoking first its `jspDestroy()` method, if present.

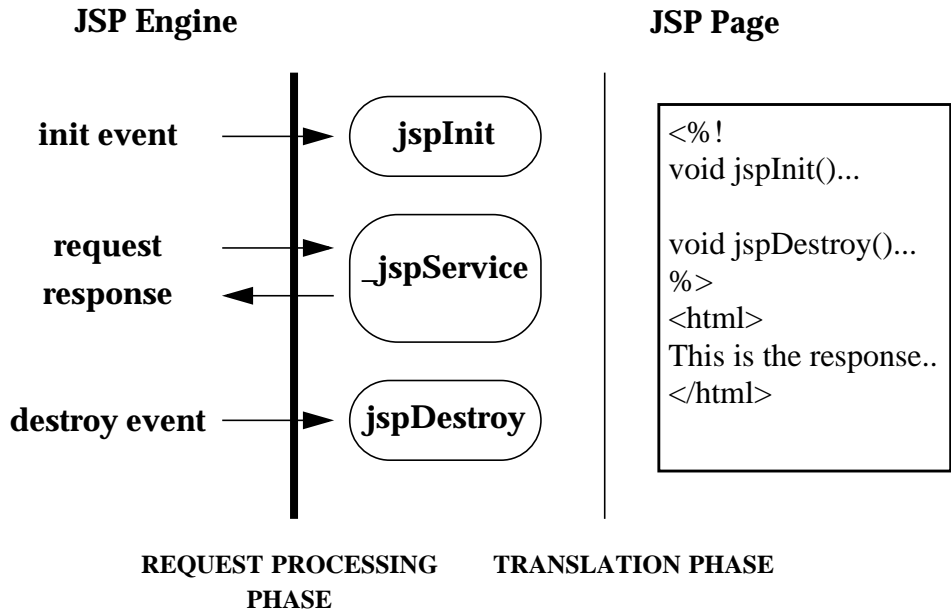
A JSP author may **not** (re)define any of the Servlet methods through a declaration scripting element.

The JSP specification reserves the semantics of methods and variables starting with `_jsp`, `jsp`, `_jspx`, or `_jspx`.

### *The HttpJspPage Interface*

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

FIGURE 4-1 Contracts between a JSP Page and a JSP Engine.



The involved contracts are shown in FIGURE 4-1. We now revisit this whole process in more detail.

---

## 4.2 JSP Page Implementation Class

The JSP engine creates a Java implementation class for each JSP page. The name of the Java implementation class is JSP engine-implementation dependent.

The creation of the Java implementation class for a JSP page may be done only by the JSP engine, or it may involve a superclass provided by the JSP author through the use of the *extends* attribute in the *jsp* directive. The *extends* mechanism is available for sophisticated users and it should be used with extreme care as it restricts what some of the decisions that a JSP Engine can take, e.g. to improve performance.

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

A JSP developer writes a JSP page expecting that the client and the server will communicate using a certain protocol. The JSP engine must then guarantee that requests from and responses to the page use that protocol. Most JSP pages use HTTP, and their implementation classes must implement the `HttpJspPage` interface (see CODE EXAMPLE 1-1 on page 87), which extends `JspPage`. If the protocol is not HTTP, then the class will implement an interface that extends `JspPage`.

## 4.2.1 API Contracts

The contract between the JSP engine and a Java class implementing a JSP page corresponds to the `Servlet` interface; refer to the Servlet specification for details.

The contract between the JSP engine and the JSP page author is described in TABLE 4-1. The responsibility for adhering to this contract rests only on the JSP Engine implementation if the JSP page does not use the `extends` attribute of the `jsp` directive; otherwise, the JSP author guarantees that the superclass given in the `extends` attribute supports this contract.

TABLE 4-1 How the JSP Engine Processes JSP Pages

Comments	Methods the JSP Engine Invokes
<p>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 <code>getServletConfig()</code> are available</p>	<pre>void <b>jspInit</b>()</pre>
<p>Method is optionally defined in JSP page. Method is invoked before destroying the page.</p>	<pre>void <b>jspDestroy</b>()</pre>
<p>Method may <b>not</b> be defined in JSP page. The JSP engine automatically generates this method, based on the contents of the JSP page. Method invoked at each client request.</p>	<pre>void <b>_jspService</b>(&lt;ServletRequestSubtype&gt;, &lt;ServletResponseSubtype&gt;) throws IOException, ServletException</pre>

## 4.2.2 Request and Response Parameters

As shown in TABLE 4-1, the methods in the contract between the JSP engine and the JSP page require request and response parameters.

The formal type of the request parameter (which this specification calls `<ServletRequestSubtype>`) is an interface that extends `javax.servlet.ServletRequest`. The interface must define a protocol-dependent request contract between the JSP engine and the class that implements the JSP page.

Likewise, the formal type of the response parameter (which this specification calls `<ServletResponseSubtype>`) is an interface that extends `javax.servlet.ServletResponse`. The interface must define a protocol-dependent response contract between the JSP engine and the class that implements the JSP page.

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

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

---

**Note** – JSP engines that conform to this specification (in both JSP implementation classes and JSP runtime environments) must implement the `request` and `response` interfaces for the HTTP protocol as described in this section.

---

### 4.2.3 Omitting the `extends` Attribute

If the `extends` attribute of the `language` directive (see Section 2.8.1, “The page Directive”) in a JSP page is not used, the JSP engine can generate any Java class that satisfies the contract described in TABLE 4-1 when it transforms the JSP page.

In the following code examples, CODE EXAMPLE 4-1 illustrates a generic HTTP superclass named `ExampleHttpSuper`. CODE EXAMPLE 4-2 shows a subclass named `_jsp1344` that extends `ExampleHttpSuper` and is the Java class generated from the JSP page. By using separate `_jsp1344` and `ExampleHttpSuper` classes, the JSP translator needs not discover if the JSP page includes a declaration with `jspInit()` or `jspDestroy()`; this simplifies very significantly the implementation.

#### CODE EXAMPLE 4-1 A Generic HTTP Superclass

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

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

abstract class ExampleHttpSuper implements HttpJspPage {
    private ServletConfig config;

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

    final public ServletConfig getServletConfig() {
        return config;
    }

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

    final public void destroy() throws ServletException {
        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 4-2 The Java Class Generated From a JSP Page

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

/**
 * An example of a class generated for a JSP.
 *
 * The name of the class is unpredictable.
 * We are assuming that this is an HTTP JSP page (like almost all are)
 */

class _jsp1344 extends ExampleHttpSuper {

    // Next code inserted directly via declarations.
    // Any of the following pieces may or not be present
    // if they are not defined here the superclass methods
    // will be used.

    public void jspInit() {...}
    public void jspDestroy() {...}

    // The next method is generated automatically by the
    // JSP processor.
    // body of JSP page

    public void _jspService(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {

        // initialization of the implicit variables

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

        // for this example, we assume a buffered directive

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

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

    }
}
```

## 4.2.4 Using the `extends` Attribute

If the JSP author uses `extends`, the generated class is identical to the one shown in CODE EXAMPLE 4-2, except that the class name is the one specified in the `extends` attribute.

The contract on the JSP Page implementation class does not change. The JSP engine 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 author that the provided superclass satisfies:

- The `service()` method of the Servlet API invokes the `_jspService()` method.
- The `init(ServletConfig)` method stores the configuration, makes it available as `getServletConfig`, then invokes `jspInit`.
- The `destroy` method invokes `jspDestroy`.

A JSP Engine may give a fatal translation error if it detects that the provided superclass does not satisfy these requirements, but most JSP Engines will not check them.

---

## 4.3 Buffering

The JSP engine buffers data (if the `jsp` directive specifies it using the `buffer` attribute) as it is sent from the server to the client. Headers are not sent to the client until the first `flush` method is invoked. Therefore, none of the operations that rely on headers, such as the `setContentType`, `redirect`, or `error` methods are valid until the `flush` method is executed and the headers are sent.

JSP 1.0 includes a class that buffers and sends output, `javax.servlet.jsp.JspWriter`. The `JspWriter` class is used in the `_jspPageService` method as in the following example:

```

import javax.servlet.jsp.JspWriter;

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

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

```

You can find the complete listing of `javax.servlet.jsp.JspWriter` in Appendix A.

With buffering turned on, you can still use a redirect method in a scriptlet in a `.jsp` file, like this:

```

<HTML>
.
.
<%
    if ....

    // out.clear() -- this is not needed
    response.redirect(someURL)
%>

```



---

# Scripting Elements Based on Java

---

This chapter describes the details of the Scripting Elements when the language directive value is “java”. The scripting language is based on the Java language (as specified by “The Java Language Specification”), but it note that there is no valid JSP page, or a subset of it, that is a valid Java program.

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

---

## 5.1 Overall Structure

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

### *Valid JSP Page*

A JSP Page is valid for a Java Platform (JRE 1.1, JRE 1.2) if and only if the JSP page implementation class defined by TABLE 5-1 (after applying all include directives), together with any other classes defined by the JSP Engine, is a valid Java program for the given Java Platform.

---

**Note** – Sun Microsystems reserves all names of the form `{_}jsp_*` and `{_}jspx_*` for the JSP specification. Names that are not defined are reserved for future expansion.

---

## Implementation Flexibility

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

TABLE 5-1 How the Java Class is Structured

SuperClass is either selected by the JSP Engine or by the JSP author via jsp directive.	<code>class _jspXXX extends SuperClass</code>
Name of class (_jspXXX) is implementation dependent.	
Optional imports clause as indicated via jsp directive	<code>imports name1, ...</code>
Start of body of JSP page implementation class	<code>{</code>
(1) Declaration Section	<code>// declarations ...</code>
signature for generated method	<code>public void _jspService(&lt;ServletRequestSubtype&gt; request,     &lt;ServletResponseSubtype&gt; response)     throws ServletException, IOException {</code>
(2) Implicit Objects Section	<code>// code that defines and initializes request, response, page,     pageContext etc.</code>
(3) Main Section	<code>// code that defines request/response mapping</code>
close of _jspService method	<code>}</code>
close of _jspXXX	<code>}</code>

---

## 5.2 Declarations Section

The declarations section correspond to the declaration elements.

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

---

## 5.3 Initialization Section

This section defines and initializes the implicit objects available to the JSP page. See Section 2.9, “Implicit Objects”.

---

## 5.4 Main Section

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

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

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

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

```
out.print(template);
```

2. A *scriptlet* is transformed into its Java statement fragment.
3. An *expression* is transformed into a Java statement to insert the value of the expression, converted to `java.lang.String` if needed, into the stream currently named by the implicit variable `out`. No additional newlines or space is included.

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

```
out.print(expression);
```

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

The semantics of the action type determine the name of the variables (usually that of the `id` attribute, if present) and their type. The only standard action in JSP 1.0 that defines objects is the `jsp:usebean` action; the name of the variable introduced is that of the `id` attribute, its type that of the `class` attribute.

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

## JSP Classes

---

---

### A.1 Package Description

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

---

### A.2 The JSP Author Contract

There are two main interfaces: *HttpJspPage* and *JspPage*. The *JspPage* interface is a partial interface in that it does not include the methods with signature `<ServletRequestSubtype>` and `<ServletResponseSubtype>`; this is because of expressive limitations in the Java type system. The *HttpJspPage* is an interface that extends the *JspPage* interface where these methods are fully qualified to the `HttpProtocol` case.

The two interfaces follow:

#### A.2.1 The *JspPage* Interface.

### CODE EXAMPLE 1-1 The JSPPage Interface

```
/**
 * This is the interface that a JSP processor-generated class must
 * satisfy
 */

package javax.servlet.jsp;
import javax.servlet.*;

interface JSPPage extends Servlet {

    /**
     * Methods that can be DEFINED BY THE JSP AUTHOR
     * either directly (via a declaration) or via an event handler
     * (in JSP 1.1)
     */

    /**
     * jsp_init() is invoked when the JSPPage is initialized.
     * At this point getServletConfig() will return the desired value
     */

    public void jspInit();

    /**
     * jsp_destroy() is invoked when the JSPPage is about to be destroyed
     */

    public void jspDestroy();

    /**
     * service is the main service entry from the superclass. It is
     * responsible from determine if the protocol is valid and to call
     * into the _jspService() method after the appropriate casting.
     */

    /**
     * _jspService corresponds to the body of the JSP page.
     * This method is defined automatically by the JSP processor
     * and should NEVER BE DEFINED BY THE JSP AUTHOR
     *
     * The specific signature depends on the protocol supported by
     * the JSP page.
     *
     * public void _jspService(<ServletRequestSubtype> request,
     *     <ServletResponseSubtype> response)
     *     throws ServletException, IOException;
     */
}

```

## A.2.2 The HttpJspPage Interface

**CODE EXAMPLE 1-1** The HttpJspPage Interface

```
/**
 * This is the interface that a JSP processor-generated class for the
 * HTTP protocol must satisfy
 */

package javax.servlet.jsp;

import javax.servlet.*;
import javax.servlet.http.*;

interface HttpJspPage extends JspPage {

    /**
     * _jspService corresponds to the body of the JSP page.
     * This method is defined automatically by the JSP processor
     * and should NEVER BE DEFINED BY THE JSP AUTHOR
     */

    public void _jspService(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException;
}
```

---

## A.3 JspWriter

The output of a JSP is written to the object that is initially referenced by the “out” implicit scripting variable. The type of this variable is `JspWriter`. A `JspWriter` is constructed by a `PageContext` object, and associated with the `PrintWriter` of the `ServletResponse`, early in the processing of the `_jspService()` method. Output written to this “initial” `JspWriter` may be buffered and subsequently, or immediately written through to the `PrintWriter`.

The `JspWriter` interface includes behaviors from `java.io.BufferedWriter` and `java.io.PrintWriter` API’s but incorporates buffering and error handling facilities for JSP.

**CODE EXAMPLE 1-1** The JspWriter interface

```
package javax.servlet.jsp;

import java.io.IOException;
```

```

/**
 * This interface emulates some of the functionality found in the
 * java.io.Writer, java.io.BufferedWriter, and java.io.PrintWriter
 * classes, however it differs in that it throws
java.io.IOException
 * from the print methods.
 *
 * @see java.io.Writer
 * @see java.io.BufferedWriter
 * @see java.io.PrintWriter
 */

public interface JspWriter {

/**
 * Write a single character. The character to be written is contained in
 * the 16 low-order bits of the given integer value; the 16 high-order bits
 * are ignored.
 *
 * Subclasses that intend to support efficient single-character output
 * should override this method.
 *
 * @exception IOException If an I/O error occurs
 */

void write(int c) throws IOException;

/**
 * Write an array of characters.
 *
 * @param cbuf Array of characters to be written
 *
 * @exception IOException If an I/O error occurs
 */

void write(char cbuf[]) throws IOException;

/**
 * Write a portion of an array of characters.
 *
 * @param cbuf Array of characters
 * @param off Offset from which to start writing characters
 * @param len Number of characters to write
 *
 * @exception IOException If an I/O error occurs
 */

void write(char[] cbuf, int off, int len) throws IOException;

/** * Write a string.
 *
 * @param str String to be written
 *

```

```

* @exception IOException If an I/O error occurs
*/

void write(String str) throws IOException;

/**
 * Write a portion of a string.
 *
 * @param str A String
 * @param off Offset from which to start writing characters
 * @param len Number of characters to write
 *
 * @exception IOException If an I/O error occurs
 */

void write(String str, int off, int len) throws IOException;

/** * Write a line separator. The line separator string is defined by the
 * system property <code>line.separator</code>, and is not necessarily a single
 * newline ('\n') character.
 *
 * @exception IOException If an I/O error occurs
 */

void newLine() throws IOException;

/**
 * Print a boolean value. The string produced by <code>{@link
 * java.lang.String#valueOf(boolean)}</code> is translated into bytes
 * according to the platform's default character encoding, and these bytes
 * are written in exactly the manner of the <code>{@link
 * #write(int)}</code> method.
 *
 * @param b The <code>boolean</code> to be printed
 * @throws java.io.IOException
 */

void print(boolean b) throws IOException;

/**
 * Print a character. The character is translated into one or more bytes
 * according to the platform's default character encoding, and these bytes
 * are written in exactly the manner of the <code>{@link
 * #write(int)}</code> method.
 *
 * @param c The <code>char</code> to be printed
 * @throws java.io.IOException
 */

void print(char c) throws IOException;

/**
 * Print an integer. The string produced by <code>{@link
 * java.lang.String#valueOf(int)}</code> is translated into bytes according

```

```

* to the platform's default character encoding, and these bytes are
* written in exactly the manner of the <code>{@link #write(int)}</code>
* method.
*
* @param      i    The <code>int</code> to be printed
* @see        java.lang.Integer#toString(int)
* @throws     java.io.IOException
*/

void print(int i) throws IOException;

/**
 * Print a long integer.  The string produced by <code>{@link
 * java.lang.String#valueOf(long)}</code> is translated into bytes
 * according to the platform's default character encoding, and these bytes
 * are written in exactly the manner of the <code>{@link #write(int)}</code>
 * method.
 *
 * @param      l    The <code>long</code> to be printed
 * @see        java.lang.Long#toString(long)
 * @throws     java.io.IOException
*/

void print(long l) throws IOException;

/**
 * Print a floating-point number.  The string produced by <code>{@link
 * java.lang.String#valueOf(float)}</code> is translated into bytes
 * according to the platform's default character encoding, and these bytes
 * are written in exactly the manner of the <code>{@link #write(int)}</code>
 * method.
 *
 * @param      f    The <code>float</code> to be printed
 * @see        java.lang.Float#toString(float)
 * @throws     java.io.IOException
*/

void print(float f) throws IOException;

/**
 * Print a double-precision floating-point number.  The string produced by
 * <code>{@link java.lang.String#valueOf(double)}</code> is translated into
 * bytes according to the platform's default character encoding, and these
 * bytes are written in exactly the manner of the <code>{@link
 * #write(int)}</code> method.
 *
 * @param      d    The <code>double</code> to be printed
 * @see        java.lang.Double#toString(double)
 * @throws     java.io.IOException
*/

void print(double d) throws IOException;

```

```

/**
 * Print an array of characters. The characters are converted into bytes
 * according to the platform's default character encoding, and these bytes
 * are written in exactly the manner of the {@link #write(int)}
 * method.
 * @param      s      The array of chars to be printed
 *
 * @throws     NullPointerException  If s is null
 * @throws     java.io.IOException
 */

void print(char s[]) throws IOException;

/**
 * Print a string. If the argument is null then the string
 * "null" is printed. Otherwise, the string's characters are
 * converted into bytes according to the platform's default character
 * encoding, and these bytes are written in exactly the manner of the
 * {@link #write(int)} method.
 *
 * @param      s      The String to be printed
 * @throws     java.io.IOException
 */

void print(String s) throws IOException;

/**
 * Print an object. The string produced by the {@link
 * java.lang.String#valueOf(Object)} method is translated into bytes
 * according to the platform's default character encoding, and these bytes
 * are written in exactly the manner of the {@link #write(int)}
 * method.
 *
 * @param      obj    The Object to be printed
 * @see        java.lang.Object#toString()
 * @throws     java.io.IOException
 */

void print(Object obj) throws IOException;

/**
 * Terminate the current line by writing the line separator string. The
 * line separator string is defined by the system property
 * line.separator, and is not necessarily a single newline
 * character ('\n').
 * @throws     java.io.IOException
 */

void println() throws IOException;

/**
 * Print a boolean value and then terminate the line. This method behaves
 * as though it invokes {@link #print(boolean)} and then
 * {@link #println()}.

```

```

* @throws          java.io.IOException
*/

void println(boolean x) throws IOException;

/**
 * Print a character and then terminate the line. This method behaves as
 * though it invokes <code>{@link #print(char)}</code> and then <code>{@link
 * #println()}</code>.
 * @throws          java.io.IOException
 */

void println(char x) throws IOException;

/**
 * Print an integer and then terminate the line. This method behaves as
 * though it invokes <code>{@link #print(int)}</code> and then <code>{@link
 * #println()}</code>.
 * @throws          java.io.IOException
 */

void println(int x) throws IOException;

/**
 * Print a long integer and then terminate the line. This method behaves
 * as though it invokes <code>{@link #print(long)}</code> and then
 * <code>{@link #println()}</code>.
 * @throws          java.io.IOException
 */

void println(long x) throws IOException;

/**
 * Print a floating-point number and then terminate the line. This method
 * behaves as though it invokes <code>{@link #print(float)}</code> and then
 * <code>{@link #println()}</code>.
 * @throws          java.io.IOException
 */

void println(float x) throws IOException;

/**
 * Print a double-precision floating-point number and then terminate the
 * line. This method behaves as though it invokes <code>{@link
 * #print(double)}</code> and then <code>{@link #println()}</code>.
 * @throws          java.io.IOException
 */

void println(double x) throws IOException;

/**
 * Print an array of characters and then terminate the line. This method
 * behaves as though it invokes <code>{@link #print(char[])</code> and then
 * <code>{@link #println()}</code>.

```

```

* @throws          java.io.IOException
*/

void println(char x[]) throws IOException;

/**
 * Print a String and then terminate the line. This method behaves as
 * though it invokes {@link #print(String)} and then
 * {@link #println()}.
 * @throws          java.io.IOException
 */

void println(String x) throws IOException;

/**
 * Print an Object and then terminate the line. This method behaves as
 * though it invokes {@link #print(Object)} and then
 * {@link #println()}.
 * @throws          java.io.IOException
 */

void println(Object x) throws IOException;
/**
 * Clear the current contents of the buffer
 *
 * @throws IOException If an I/O error occurs
 * @throws IllegalStateExceptionIf unbuffered
 */

void clear() throws IOException;

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

void flush() throws IOException;

/**
 * Close the stream, flushing it first. Once a stream has been closed,
 * further write() or flush() invocations will cause an IOException to be
 * thrown. Closing a previously-closed stream, however, has no effect.
 *
 * @exception IOException If an I/O error occurs
 */

void close() throws IOException;

/**

```

```

* @return the size of the buffer in bytes, or 0 is unbuffered.
*/

int getBufferSize();

/**
* @return the number of bytes unused in the buffer
*/

int getRemaining();

/**
* @return if this JspWriter is auto flushing or throwing IOExceptions on buffer
overflow conditions
*/

boolean isAutoFlush();
}

```

---

## A.4 PageContext

A `PageContext` object is initialized by a JSP implementation early on in the processing of the `_jspService()` method. The `PageContext` implementation itself is implementation dependent, and is obtained via a creation method on the `JspFactory`.

The `PageContext` provides a number of facilities, including:

- a single API that manages the operations available over various scope namespaces (`page`, `request`, `session`, `application`) such as `setAttribute()`, `getAttribute()` and `removeAttribute()`, etc.
- a mechanism for obtaining a platform dependent implementation of the `JspWriter` that is assigned to the “out” implicit scripting variable.
- a number of simple convenience *getter* API’s for obtaining references to various request-time objects.
- mechanisms to *forward* or *include* the current request being processed to other components in the application

### CODE EXAMPLE 1-1 The `PageContext` abstract class

```

package javax.servlet.jsp;

import java.io.IOException;
import java.util.Enumeration;
import java.util.Hashtable;

```

```

import java.util.NoSuchElementException;

import javax.servlet.Servlet;
import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

/**
 * <p>
 * An instance of an implementation dependent subclass of this abstract base
 * class is created by a JSP implementation class at the beginning of it's
 * <code> _jspService() </code> method via an implementation default
 * <code> JspFactory </code>, as follows:
 * </p>
 * <p>
 * <p>
 * <code>
 * <pre>
 * public void _jspService(HttpServletRequest request,
 *                          HttpServletResponse response)
 *     throws IOException, ServletException {
 *
 *     JspFactory factory = JspFactory.getDefaultFactory();
 *     PageContext pageContext = factory.createPageContext(
 *         this,
 *         request,
 *         response,
 *         true, // needs a session
 *         PageContext.DEFAULT_BUFFER,
 *         true // autoflush=true
 *     );
 *
 *     HttpSession session = pageContext.getSession();
 *     JspWriter out = pageContext.getInitialOut();
 *     Object page = this;
 *
 *     // body of JSP here ...
 *
 * }
 * </pre>
 * </code>
 * </p>
 * <p>
 * The <code> PageContext </code> class is an abstract class, designed to be
 * extended to provide implementation dependent implementations thereof, by
 * conformant JSP engine runtime environments.
 * </p>
 * <p>
 * The PageContext provides a number of facilities to the page author and

```

```

* page implementor, including:
* <td>
* <li>a single API to manage the various scoped namespaces
* <li>a number of convenience API's to access various public objects
* <li>a mechanism to obtain the JspWriter for output
* <li>a mechanism to manage session usage by the page
* <li>a mechanism to expose page directive attributes to the scripting
environment
* <li>mechanisms to forward or include the current request to other active
components in the application
* </td>
* </p>
*/

abstract public class PageContext {

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

public static final int PAGE_SCOPE= 1;

/**
* request scope: the named reference remains available from the ServletRequest
* associated with the Servlet that until the current request is completed.
*/

public static final int REQUEST_SCOPE= 2;

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

public static final int SESSION_SCOPE= 3;

/**
* application scope: named reference remains available in the
* ServletContext until it is reclaimed.
*/

public static final int APPLICATION_SCOPE= 4;

/**
* name used to store the Servlet in this PageContext's nametables
*/

public static final StringPAGE= "javax.servlet.jsp:JspPage";

/**
* name used to store this PageContext in it's own name tables

```

```

*/

public static final StringPAGECONTEXT= "javax.servlet.jsp:JspPageContext";

/**
 * name used to store ServletRequest in PageContext name table
 */

public static final StringREQUEST= "javax.servlet.jsp:JspRequest";

/**
 * name used to store ServletResponse in PageContext name table
 */

public static final StringRESPONSE= "javax.servlet.jsp:JspResponse";

/**
 * name used to store ServletConfig in PageContext name table
 */

public static final StringCONFIG= "javax.servlet.jsp:JspConfig";

/**
 * name used to store HttpSession in PageContext name table
 */

public static final StringSESSION= "javax.servlet.jsp:JspSession";
/**
 * name used to store current JspWriter in PageContext name table
 */

public static final StringOUT= "javax.servlet.jsp:JspOut";

/**
 * name used to store ServletContext in PageContext name table
 */

public static final StringAPPCONTEXT= "javax.servlet.jsp:JspAppContext";

/**
 * name used to store uncaught exception in ServletRequest attribute list and
PageContext name table
 */

public static final StringEXCEPTION= "javax.servlet.jsp:JspException";

/**
 * constant indicating that the page is not buffering output
 */

public static final intNO_BUFFER = 0;

/**

```

```

* constant indicating that the page wants the implementation default buffer
size
*/

public static final int DEFAULT_BUFFER = -1;

        /**
* @return the object named or null if not found
*/

public Object getAttribute(String name);
/**
* register the name and object specified with page scope semantics
*
* @throws NullPointerException if the name or object is null
*/

public void setAttribute(String name, Object attribute);
/**
* register the name and object specified with appropriate scope semantics
*
* @throws NullPointerException if the name or object is null
* @throws IllegalArgumentException if the scope is invalid
*
*/

public void setAttributeWithScope(String name, Object o, int scope);

/**
* remove the object reference associated with the specified name
*/

public void removeAttribute(String name);
/**
* remove the object reference associated with the specified name
*/

public void removeAttributeInScope(String name, int scope);
/**
* @return the scope of the object associated with the name specified or 0
*/

public int getAttributesScope(String name);
/**
* @return an enumeration of name/object associations in the specified scope
*/

public Enumeration getAttributesInScope(int scope);

/**
* @return if output is buffered.
*/

```

```

public boolean isOutBuffered();

/**
 * @return if the JspWriter is auto-flushing to the ServletResponse PrintWriter
 * or if it is throwing an IOException on buffer overflow.
 */

public boolean isAutoFlush();

/**
 * @return the buffer size of the output JspWriter
 */

public int getOutBufferSize();
/**
 * @return the initial JspWriter stream being used for client response
 */

public JspWriter getInitialOut();
/**
 * @return the current JspWriter stream being used for client response
 */

public JspWriter getOut();

/**
 * @return if the Servlet associated with this PageContext needs an HttpSession
 */

public boolean getNeedsSession();
/**
 * @return the HttpSession for this PageContext or null
 */

public HttpSession getSession();

/**
 * @return the Servlet associated with this PageContext
 */

public Servlet getServlet();
/**
 * @return the ServletConfig for this PageContext
 */

public ServletConfig getServletConfig();
/**
 * @return the ServletContext for this PageContext
 */

public ServletContext getServletContext();
/**
 * @return The ServletRequest for this PageContext
 */

```

```

public ServletRequest getServletRequest();
/**
 * @return the ServletResponse for this PageContext
 */

public ServletResponse getServletResponse();
/**
 * @return the JSPFactory that created this instance
 */

public JspFactory getFactory();

/**
 * @return any exception passed to this as an errorpage
 */

public Exception getException();
/**
 * <p>
 * This method is used to re-direct, or "forward" the current ServletRequest
 and ServletResponse to another active component in the application.
 * </p>
 * <p>
 * If the <I> relativeUrlPath </I> begins with a "/" then the URL specified
 * is calculated relative to the DOCROOT of the <code> ServletContext </code>
 * for this JSP. If the path does not begin with a "/" then the URL
 * specified is calculated relative to the URL of the request that was
 * mapped to the calling JSP.
 * </p>
 * <p>
 * It is only valid to call this method from a <code> Thread </code>
 * executing within a <code> _jspService(...) </code> method of a JSP.
 * </p>
 * <p>
 * Once this method has been called successfully, it is illegal for the
 * calling <code> Thread </code> to attempt to modify the <code>
 * ServletResponse </code> object. Any such attempt to do so, shall result
 * in undefined behavior. Typically, callers immediately return from
 * <code> _jspService(...) </code> after calling this method.
 * </p>
 *
 * @param relativeUrlPath specifies the relative URL path to the target resource
 as described above
 *
 * @throws ServletException
 * @throws IOException
 *
 * @throws IllegalArgumentException if target resource URL is unresolvable
 * @throws IllegalStateException if <code> ServletResponse </code> is not in a
 state where a forward can be performed
 * @throws SecurityException if target resource cannot be accessed by caller
 */

```

```
abstract public void forward(String relativeUrlPath) throws ServletException,
IOException;
```

```
/**
 * <p>
 * Causes the resource specified to be processed as part of the current
 * ServletRequest and ServletResponse being processed by the calling Thread.
 * The output of the target resources processing of the request is written
 * the the JspWriter specified.
 * </p>
 * <p>
 * If the <I> relativeUrlPath </I> begins with a "/" then the URL specified
 * is calculated relative to the DOCROOT of the <code> ServletContext </code>
 * for this JSP. If the path does not begin with a "/" then the URL
 * specified is calculated relative to the URL of the request that was
 * mapped to the calling JSP.
 * </p>
 * <p>
 * It is only valid to call this method from a <code> Thread </code>
 * executing within a <code> _jspService(...) </code> method of a JSP.
 * </p>
 *
 * @param relativeUrlPath specifies the relative URL path to the target resource
 * to be included
 * @param out specifies the JspWriter to "include" the output into.
 *
 * @throws ServletException
 * @throws IOException
 *
 * @throws IllegalArgumentException if the target resource URL is unresolvable
 * @throws SecurityException if target resource cannot be accessed by caller
 *
 */
```

```
abstract public void include(String relativeUrlPath, JspWriter out) throws
ServletException, IOException;
```

```
/**
 * implementation private method: for use by subclasses only
 *
 * called by constructor(s) to create the initial out JspWriter
 * must be implemented by concrete subclass(es).
 *
 * @param buffersize size of buffer in bytes, unbuffered, or default
 * @param autoflush
 *
 * @throws IOException
 * @throws IllegalArgumentException
 *
 */
```

```
abstract protected JspWriter createInitialOut(int buffersize, boolean
```

```
autoflush) throws IOException, IllegalArgumentException;

}
```

---

## A.5 JspFactory

The `JspFactory` provides a mechanism to instantiate platform dependent objects in a platform independent manner. In this version of the specification the `PageContext` class is the only implementation dependent class that can be created from the factory.

Typically at initialization time, a JSP engine will call the static `setDefaultFactory()` method in order to register it's own factory implementation. JSP implementation classes will use the `getDefaultFactory()` method in order to obtain this factory and use it to construct `PageContext` instances in order to process client requests.

**CODE EXAMPLE 1-1** The `JspFactory` abstract class

```
package javax.servlet.jsp;

import javax.servlet.Servlet;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

import javax.servlet.jsp.PageContext;

/**
 * <p>
 * The JspFactory is an abstract class that defines a number of factory
 * methods available to a JSP page at runtime for the purposes of creating
 * instances of various interfaces and classes used to support the JSP
 * implementation.
 * </p>
 * <p>
 * A conformant JSP Engine implementation will, during it's initialization
 * instantiate an implementation dependent subclass of this class, and make
 * it globally available for use by JSP implementation classes by registering
 * the instance created with this class via the static <code>
 * setDefaultFactory() </code> method.
 * </p>
 */

public abstract class JspFactory {

    private static JspFactory deflt = null;

    /**
```

```

* <p>
* set the default factory for this implementation. It is illegal for
* any principal other than the JSP Engine runtime to call this method.
* </p>
*
* @param default    The default factory implementation
*/

public static synchronized void setDefaultFactory(JspFactory deflt) {
    JspFactory.deflt = deflt;
}

/**
* @return the default factory for this implementation
*/

public static synchronized JspFactory getDefaultFactory() {
    return deflt;
}

/**
* <p>
* Creates an instance of an implementation dependent class that implements
* the javax.servlet.jsp.PageContext interface for the Servlet, ServletRequest
* </p>
*
* @param servlet    The Servlet for this JSP
* @param request    The current ServletRequest for this JSP
* @param response   The current ServletResponse for this JSP
* @param needsSession true if the JSP participates in a session
* @param buffer     size of buffer in bytes, NO_BUFFER if no buffer,
DEFAULT_BUFFER if implementation default.
* @param autoflush  should the buffer autoflush to the output stream on
buffer overflow, or throw an IOException?
*
* @return the page context
*/

public abstract PageContext createPageContext(Servlet s, ServletRequest req,
ServletResponse resp, boolean needsSession, int buffer, boolean autoflush);
}

```



## Java<sup>TM</sup> Servlet 2.1 clarification(s)

---

This appendix provides clarifications on several portions of the Java Servlet 2.1 specification that are necessary for the JSP 1.0 specification.

---

**Note** – Some details are still being discussed with the Java Servlet expert group; a final clarification that will apply to both Servlet 2.1 and JSP 1.0 will be included in the final JSP 1.0 specification and will be referred to from the Java Servlet document site.

---

### B.1 Relative URL interpretation

A number of API's in the Servlet 2.1 specification take a URL as a parameter; in particular:

- `ServletContext.getRealPath()`
- `ServletContext.getResource()`
- `ServletContext.getResourceAsStream()`
- `ServletContext.getRequestDispatcher()`

All of these methods take a relative URL path. If this relative path begins with a '/' then the path is calculated relative to the `DOCROOT` of the `ServletContext`. If no initial '/' is present, then the path is calculated relative to that of the calling JSP.

The `ServletContext.getContext()` method also takes a relative URL path. This URL is relative to the `DOCROOT` of the "root" `ServletContext` for the runtime, that is the `ServletContext` mapped to '/', the `DOCROOT` of the web or application server runtime environment.

Note that `ServletContext.getRealPath("")` shall return the the corresponding physical path to the `ServletContext`'s virtual root (`DOCROOT`).

---

## B.2 Sharing Session State

The `javax.servlet.http.HttpSession` interface allows, via the `setValue()`, `getValue()`, `removeValue()`, and `getValueNames()` methods, the ability to share Java objects between components in the same session. For application security and integrity this shared state is private to components that share the same `ServletContext`, that is, it is not possible to share state between components with different `ServletContext` references even they may be in the same session.

---

## B.3 Access Control

The Servlet API permits access to various resources on the server via the following APIs:

- `ServletContext.getResource()`
- `ServletContext.getResourceAsString()`
- `ServletContext.getContext()`
- `ServletContext.getMimeType()`
- `ServletContext.getRealPath()`
- `ServletContext.getRequestDispatcher()`

In more sophisticated web and application runtime environments (especially those that will implement Java<sup>™</sup> 2 Enterprise Edition) access to resources on the server may be protected and thus require certain user privilege to gain access to them. In such environments, when access is attempted using the methods listed above, a compliant implementation may throw a `SecurityException` (subclass) to indicate that the caller had insufficient privilege to access the underlying resource that was the subject of the operation.

This optional semantic may become a requirement in a future version of this, or the Servlet specification.

---

## B.4 Path mapping

The Servlet API provides a number of APIs to interrogate the URL path of a `Servlet`. Since a JSP implements `Servlet` it has access to these APIs.

When called from a JSP Servlet the following APIs exhibit the semantics described below:

- `getRequestURI()`  
returns the root relative part of the URL that the client requested (that is an unmapped URL).
- `HttpServletRequest.getPathInfo()`  
returns the same as `getRequestURI()`.
- `getPathTranslated()`  
returns the actual, physical, path of the JSP
- `getServletPath()`  
returns the Servlet path for the JSP

---

**Note** – Path details are still being coordinated with the Java Servlet expert group.

---



# Change History

---

---

## C.1 Changes Between 0.92 and 1.0

A fair amount of changes have happened between 0.92 and 1.0. Some of the 0.92 functionality has been delayed into a separate release 1.1 that is being developed in parallel.

Most of the specification has been rewritten or changed significantly.

### C.1.1 Normalized Usage

### C.1.2 Changes

- SSI have been replaced with a `<%@ include` directive
- Tags are case sensitive, as in XML, and XHTML.
- Standard tags now follow the mixed-case convention from the Java Platform.
- `jsp:setProperty`, and `jsp:getProperty` have been defined.
- `<SCRIPT> </SCRIPT>` is replaced by `<%! ... %>`.
- Normalized usage in several places. We now use *elements*, *directives*, and *actions*. *Tags* now just mean the actual tag in the elements, as in *start*, *end*, and *empty* tags. The use of the terms *elements* and *tags* is thus consistent with that in HTML, XML, SGML, etc.

## C.1.3 Removals

The following features have been removed.

- NCSA-style SSIs are no longer mentioned explicitly in the specification.

## C.1.4 Postponed for Evaluation in Future Releases

The following features have been postponed for evaluation in future releases:

- Application (*.jsa*) files and corresponding events.
- Action elements that affect control flow (like LOOP, ITERATE, INCLUDEIF, EXCLUDEIF) have been postponed until when a Tag Extension mechanism is made available that will enable a consistent implementation of these and other features.
- A proposal for hierarchical naming to be used in the above elements.
- The `processRequest` machinery in USEBEAN.

## C.1.5 Additions

- Request-time attribute evaluation has been added.
- The semantics of a JSP page has been defined more formally.
- A `jsp:request` action element has been added to provide run-time forward and include capabilities of active resources (based on RequestDispatcher).
- A `jsp:include` action element has been added to provide run-time inclusion of static resources.
- Buffering has been added.
- The `page` directive collects several attributes; within this, the `extends` attribute corresponds to functionality that had been removed in 0.92.
- `jsp:plugin` has been defined.
- An equivalent XML document for a JSP page has been provided.

---

## C.2 Changes between 0.91 and 0.92

The 0.91 specification was the first formal public specification draft of JavaServer Pages. The specification changed substantially between 0.91 and 0.92

## Future Directions

---

This appendix provides some tentative directions for future releases of the JavaServer Pages™ specification. Information in this appendix is non-normative.

---

### D.1 JSP 1.1

JSP 1.1 is a release that is being developed in parallel with JSP 1.0. JSP 1.1 will be part of Java™ 2 Enterprise Edition 1.0.

#### D.1.1 Optional Features Become Mandatory

The optional features in JSP1.0 will be labelled mandatory in JSP 1.1. These features are:

- Request-time attribute expressions.
- All custom tags have to be expressed using a portable tag extension library.

This second feature depends on a tag extension mechanism that is being considered for JSP 1.1.

#### D.1.2 Tag Extension Mechanism

A portable tag extension mechanism is being considered for JSP 1.1. This mechanism permits the description of tags that can be used from any JSP page.

Such a mechanism may also include a way to provide a visual component that can be used, in a portable manner, to simplify the authoring of elements from the library.

## D.1.3 Additional Features

JSP 1.1 will likely include:

- Support for Application, including global state
- Support for event handlers.

Some other features are useful for JSP 1.1 on their own but are also intended to integrate smoothly within J2EE 1.0. JSP 1.1 is likely to include:

- Packaging information.
- Deployment information.

## D.1.4 Support for J2EE 1.0

JSP 1.1 will likely provide for integration with J2EE 1.0 for the following concepts:

- Security.
- Transactions.
- Session state.