JavaServer Pages™ Standard Tag Library

Version 1.0

Pierre Delisle, editor

Please send comments to jsr-52-comments@jcp.org

JavaServer PagesTM Standard Tag Library (JSTL) Specification ("Specification")
Version: 1.0
Status: FCS
Release: June 11, 2002

REPORT

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

(LFI#113300/Form ID#011801)

Contents

Preface

This is the JavaServer Pages™ Standard Tag Library 1.0 (JSTL 1.0) specification, developed by the JSR-52 expert group under the Java Community Process. See http://www.jcp.org.

Related Documentation

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

JavaServer Pages (JSP)	http://java.sun.com/jsp
Java Servlet Technology	http://java.sun.com/servlet
Java 2 Platform, Standard Edition	http://java.sun.com/j2se
Java 2 Platform, Enterprise Edition	http://java.sun.com/j2ee
JavaBeans	http://java.sun.com/beans
JDBC	http://java.sun.com/jdbc
Java Technology and XML	http://java.sun.com/xml
XPath specification	http://www.w3.org/TR/xpath
XML home page at W3C	http://www.w3.org/XML
HTML home page at W3C	http://www.w3.org/MarkUp
XML.org home page	http://www.xml.org

Typographical Conventions

Font Style	Uses
Italic	Emphasis, definition of term.
`Monospace`	Syntax, code examples, attribute names, Java language types, API, enumerated attribute values.

Acknowledgments

The JavaServer Pages™ Standard Tag Library (JSTL) specification is the result of collaborative work involving many individuals, all driven by a common goal of designing the best libraries possible for the JSP page author community.

We would like to thank all members of the JSR-52 expert group: Nathan Abramson, Shawn Bayern, Hans Bergsten, Paul Bonfanti, Vince Bonfanti, David Brown, Larry Cable, Tim Dawson, Morgan Delagrange, Bob Foster, David Geary, Scott Hasse, Hal Hildebrand, Jason Hunter, Serge Knystautas, Mark Kolb, Wellington Lacerda, Jan Luehe, Geir Magnusson Jr., Dan Malks, Craig McClanahan, Richard Morgan, Glenn Nielsen, Rickard Oberg, Joseph B. Ottinger, Eduardo Pelegri-Llopart, Sam Pullara, Tom Reilly, Brian Robinson, Russ Ryan, Pasi Salminen, Steven Sargent, Allan Scott, Virgil Sealy, Magnus Stenman, Gael Stevens, James Strachan, Christine Tomlinson, Norbert von Truchsess, Keyton Weissinger, Clement Wong, Alex Yiu.

This specification was first initiated by Eduardo Pelegri-Llopart. Eduardo's leadership in making the Java platform the best technology available for the web layer has been key in shaping the vision behind the standard tag library.

Shawn Bayern and Hans Bergsten deserve special credit for being actively involved in all design issues of this specification. Their vast expertise and commitment to excellence has had a profound impact in every single aspect of this specification. *Mille mercis Shawn et Hans!* Don't know how we would have done it without you two.

Many thanks to Jan Luehe for taking ownership of the internationalization and formatting chapters of this specification on short notice, and doing an incredible job.

Comments

We are interested in improving this specification and welcome your comments and suggestions. You can email your comments to us at:

```
jsr-52-comments@jcp.org
```

Introduction

This is the JavaServer Pages™ Standard Tag Library 1.0 (JSTL 1.0) specification, developed by the JSR-52 expert group under the Java Community Process (http://www.jcp.org).

1.1 Goals

The ultimate goal of JSTL is to help simplify JavaServer™ Pages (JSP™) page authors' lives.

A page author is someone who is responsible for the design of a web application's presentation layer using JSP pages. Many page authors are not fluent in any programming language.

One of the main difficulties a page author is faced with is the need to use a scripting language (the default being the Java programming language) to manipulate the dynamic data within a JSP page. Unfortunately, page authors often see scripting languages as complex and not very well adapted to their needs.

A key aspect of JSTL is an expression language (EL) designed specifically for the page author (see Section 1.3). In addition, JSTL has:

■ Expression language support actions

These actions allow a page author to easily display expressions in the expression language, as well as to easily set the value of JSP scoped attributes.

■ Control flow actions

Tag-based control flow structures (conditionals, iterators), which are more natural to page authors.

■ Tag library validators (TLVs)

TLVs allow projects to only allow specific tag libraries, as well as enforce JSP coding styles that are free of scripting elements.

The other key aspect of JSTL is that it provides standard actions for functionality most often needed by page authors. These cover the following topics:

■ Accessing URL-based resources

■ Internationalization (i18n) and text formatting

■ Relational database access (SQL)

■ XML processing

1.2 Multiple Tag Libraries

A tag library is a collection of actions that encapsulates functionality to be used from within a JSP page. JSTL includes a wide variety of actions that naturally fit into discrete functional areas. This is why JSTL, although referred to as the standard tag library (singular), is exposed via multiple tag libraries to clearly identify the functional areas it covers, as well as to give each area its own namespace. The tables below lists these functional areas along with the URIs used to reference the libraries. The tables also show the prefixes used in this specification (although page authors are free to use any prefix they want). The fact that two versions of the tag libraries are supported in JSTL 1.0 (EL- and RT- based versions) is explained in Section 1.3.

EL-based Tag Libraries

Functional Area	URI	Prefix
core	`http://java.sun.com/jstl/core`	c
XML processing	`http://java.sun.com/jstl/xml`	x
I18N capable formatting	`http://java.sun.com/jstl/fmt`	fmt
relational db access (SQL)	`http://java.sun.com/jstl/sql`	sql

RT-based Tag Libraries

Functional Area	URI	Prefix
core	`http://java.sun.com/jstl/core_rt`	c_rt
XML processing	`http://java.sun.com/jstl/xml_rt`	x_rt
I18N capable formatting	`http://java.sun.com/jstl/fmt_rt`	fmt_rt
relational db access (SQL)	`http://java.sun.com/jstl/sql_rt`	sql_rt

1.3 Expression Language

A key contribution of JSTL is the support for an expression language (EL). An EL leverages the fact that JSP scoped attributes as well as request parameters are the privileged way to communicate information to JSP pages. It makes it possible to easily access application data and manipulate it in simple ways without having to use scriptlets or request-time expression values.

For example, using the expression language, this conditional action tests whether or not the country of a customer is Greece.

```
<c:if test="${customer.address.country == 'Greece'}">
    ...
</c:if>
```

The specification of an expression language is ultimately the responsibility of the JSR-152 expert group (JavaServer Pages version 1.3). The JSR-52 (JSTL) and JSR-152 expert groups have worked closely together on a specification of the expression language that meets the requirements identified in JSR-52. Given that JSTL 1.0 will be finalized prior to JSP 1.3, the goal has been to ensure that the current specification of the EL will be compatible with the one that will make it in the final draft of JSR-152. The plan is to then have a maintenance release of JSTL that will be based on the official expression language specified in the final draft of JSR-152.

Supporting the EL in a JSP 1.2 environment

There were quite a few issues involved with the support of an expression language within JSTL given the constraint that it had to work with JSP 1.2 containers. In order to be able to support both the scripting (rtexprvalues) and the EL (elexprvalues) worlds, we came up with the concept of twin tag libraries: one that supports the expression language, and one that supports request time expression values. The assumption is that people will mostly use the EL-based tag libraries, while it is still possible for page authors that prefer scripting to use JSTL with rtexprvalues (provides benefits for type safety and performance) via the RT-based tag libraries. The URIs of the RT-based tag libraries simply have the "_rt" suffix appended. It is allowed to mix both EL- and RT- based actions as shown in the following example:

```
<fmt:message key="attendanceAgeAverage">
    <fmt:param value="${totalParticipants}">
    <fmt_rt:param value="<%=Stats.ageAverage(participants)%>"/>
</fmt:message>
```

For consistency, EL-based tag library prefixes are used in this specification for all references to JSTL actions. All examples also make use of the EL-based tag libraries.

1.4   Container Requirement

JSTL requires a JSP 1.2 web container.

Conventions

This chapter describes the conventions used in this specification.

2.1 How Actions are Documented

JSTL actions are grouped according to their functionality. These functional groups of actions are documented in their own chapter using the following structure:

- Motivation

  Describes the motivation for standardizing the actions.

- Overview

  Provides an overview of the capabilities provided by the actions. Sample code featuring these actions in their most common use cases is also provided.

- One section per action, with the following structure:

  - Name

    For consistency, EL-based tag library prefixes are used in this specification for all references to JSTL actions (e.g.: <c:if> instead of <if> or <c_rt:if>).

  - Short Description

  - Syntax

    The syntax notation is described in Section 2.1.2.

  - Body Content

    This section specifies which type of body content is supported by the action. As defined by the JSP specification, the body content type can be one of `empty`, `JSP`, or `tagdependent`. The section also specifies if the body content is processed by the action or is simply ignored by the action and just written to the current `JspWriter`. If the body content is processed, information is given on whether or not the body content is trimmed before the action begins processing it.

  - Attributes

Details in Section 2.1.1 below.

- Constraints

  List of additional constraints enforced by the action.

- Null & Error Handling

  Details on how null and empty values are processed, as well as on exceptions thrown by the action.

- Description

  This section provides more details on the action.

- Other sections

  Other sections related to the group of actions described in the chapter may exist. These include sections on interfaces and classes exposed by these actions.

2.1.1   Attributes

For each attribute, the following information is given: name, dynamic behavior, type, and description.

The `rtexprvalue` element defined in a TLD is covered in this specification with the column titled "Dynamic" that captures the dynamic behavior of an attribute for both the EL and RT library versions. This column can be either true or false. Static string values are always valid whether "Dynamic" is true or false. A false value in the dynamic column means that only a static string value can be specified for the attribute. The meaning associated with a true value depends on the library used. For EL, this means that EL expressions are allowed; for RT it means that rtexprvalues are allowed. In both cases, the result of evaluating the expression must yield an object whose type is compatible with the one specified in the table.

2.1.2 Syntax Notation

`[...]`	What is inside the square brackets is optional
`{option1|option2|option3|...}`	Only one of the given options can be selected
`value`	The default value

For example, in the syntax below:

```
<c:set var="varName" [scope="{page|request|session|application}"]
    value="value"/>
```

the attribute `scope` is optional. If it is specified, its value must be one of `page`, `request`, `session`, or `application`. The default value is `page`.

2.2 Scoped Variables

Actions usually collaborate with their environment in implicit or explicit ways, or both.

Implicit collaboration is often done via a well defined interface that allows nested tags to work seamlessly with the ancestor tag exposing that interface. The JSTL iterator tags support this mode of collaboration.

Explicit collaboration happens when a tag explicitly exposes information to its environment. Traditionally, this has been done by exposing a scripting variable with a value assigned from a JSP scoped attribute (which was saved by the tag handler). Because JSTL supports an expression language, the need for scripting variables is significantly reduced. This is why all the JSTL tags expose information only as JSP scoped attributes (no scripting variable exposed). These exported JSP scoped attributes are referred to as *scoped variables* in this specification; this helps in preventing too much overloading of the term "attribute".

2.2.1   var and scope

The convention is to use the name var for attributes that export information. For example, the <c:forEach> action exposes the current item of the customer collection it is iterating over in the following way:

```
<c:forEach var="customer" items="${customers}">
    Current customer is <c:out value="${customer}"/>
</c:forEach>
```

It is important to note that a name different than id was selected to stress the fact that only a scoped variable (JSP scoped attribute) is exposed, without any scripting variable.

If the scoped variable has at-end visibility (see Section 2.2.2), the convention also establishes the attribute scope to set the scope of the scoped variable.

The scope attribute has the semantics defined in the JSP specification, and takes the same values as the ones allowed in the <jsp:useBean> action; i.e. page, request, session, application. If no value is specified for scope, page scope is the default unless otherwise specified.

It is also important to note, as per the JSP specification, that specifying "session" scope is only allowed if the page has sessions enabled.

If an action exposes more than one scoped variable, the main one uses attribute names var and scope, while secondary ones have a suffix added for unique identification. For example, in the <c:forEach> action, the var attribute exposes the current item of the iteration (main variable exposed by the action), while the varStatus attribute exposes the current status of the iteration (secondary variable).

2.2.2   Visibility

Scoped variables exported by JSTL actions are categorized as either nested or at-end.

Nested scoped variables are only visible within the body of the action and are stored in "page" scope[1]. The action must create the variable according to the semantics of PageContext.setAttribute(varName, PAGE_SCOPE), and it must remove it at the end of the action according to the semantics of PageContext.removeAttribute(varName, PAGE_SCOPE).[2]

At-end scoped variables are only visible at the end of the action. Their lifecycle is the one associated with their associated scope.

---

1. Since nested scoped variables are always saved in page scope, no scope attribute is associated with them.

In this specification, scoped variables exposed by actions are considered at-end by default. If a scoped variable is nested, it will be explicitly stated.

2.3   Static vs Dynamic Attribute Values

Except for the two exceptions described below, attribute values of JSTL actions can always be specified dynamically (see Section 2.1.1); i.e. via a request-time expression value (in the scripting language of the page), or via an expression language expression value (in the JSTL expression language).

The first exception to this convention is for the `select` attribute of XML actions. This attribute is reserved in JSTL to specify a `String` literal that represents an expression in the XPath language.

The second exception is for attributes that define the name and scope of scoped variables (as introduced in Section 2.1.1) exported by JSTL actions.

Restricting these attributes to static values should benefit development tools, without any impediment to page authors.

2.4   White Spaces

Following the JSP specification (as well as the XML and XSLT specifications), whitespace characters are `#x20, #x9, #xD,` or `#xA.`

2.5   Body Content

If an action accepts a body content, an empty body is always valid, unless explicitly stated otherwise.

If the body content is used to set the value of an attribute, then an empty body content sets the attribute value to an empty string.

2. It is important to note that the JSP specification says that "A name should refer to a unique object at all points in the execution, that is all the different scopes really should behave as a single name space." The JSP specification also says that "A JSP container implementation may or may not enforce this rule explicitly due to performance reasons". Because of this, if a scoped variable with the same name as a nested variable already exists in a scope other than 'page', exactly what happens to that scoped variable depends on how the JSP container has been implemented. To comply with the JSP specification, and to avoid non-portable behavior, page authors should therefore avoid using the same name in different scopes.

If a body content is trimmed prior to being processed by the action, it is trimmed as defined in method `trim()` of the class `java.lang.String`.

2.6 Naming

JSTL adopts capitalization conventions of Java variables for compound words in action and attribute names. Recommended tag prefixes are kept lowercase. Thus, we have <sql:transaction> and <c:forEach>, as well as attributes such as `xmlUrl` and `varDom`.

In some cases, attribute names for JSTL actions carry conventional meanings. For instance, Section 2.2.1 discussed the `var` and `scope` attibutes. Section 11.1.5 discusses the `select` attribute used in JSTL's XML-processing tag library.

2.7 Errors and Exceptions

All syntax errors (as defined in the syntax section of each action, as well as the syntax of EL expressions as defined in Appendix A) must be reported at translation time.

Constraints, as defined in the constraints section of each action, must also be reported at translation time unless they operate on a dynamic attribute value, in which case errors are reported at runtime.

For the RT-based libraries, the conversion from a `String` value to the expected type of an attribute is handled according to the rules defined in the JSP specification.

Since it is hard for a page author to deal with exceptions, JSTL tries to avoid as many exception cases as possible, without causing other problems.

For instance, if <c:forEach> were to throw an exception when given a null value for the attribute `items`, it would be impossible to easily loop over a possibly missing string array that represents check-box selection in an HTML form (retrieved with an EL expression like `${paramValues.selections}`). A better choice is to do nothing in this case.

The conventions used in JSTL with respect to errors and exceptions are as follows:

- `scope`
  - Invalid value – translation time validation error
- `var`

- Empty – translation time validation error
- Dynamic attributes *with* a fixed set of valid String values:
    - null – use the default value
      A null value can therefore be used to dynamically (e.g. by request parameter), turn on or off special features without too much work.
    - Invalid value – throw an exception
      If a value is provided but is not valid, it's likely a typo or another mistake.
- Dynamic attributes *without* a fixed set of valid values:

  The rules below assume that if the type of the value does not match the expected type, the EL will have applied coercion rules to try to accomodate the input value. Moreover, if the expected type is one of the types handled by the EL coercion rules, the EL will in most cases coerce null to an approriate value. For instance, if the expected type is a `Number`, the EL will coerce a null value to 0, if it's `Boolean` it will be coerced to false.
    - null – behavior specific to the action
      If this rule is applied, it's because the EL could not coerce the null into an appropriate default value. It is therefore up to the action to deal with the null value and is documented in the "Null & Error Handling" section of the action.
    - Invalid type – throw an exception
    - Invalid value – throw an exception
- Exceptions caused by the body content:

  Always propagate, possibly after handling them (e.g. <sql:transaction>).
- Exceptions caused by the action itself:

  Always propagate, possibly after handling them.
- Exceptions caused by the EL:

  Always propagate.
- Exceptions caused by XPath:

  Always propagate.

Page authors may catch an exception using <c:catch>, which exposes the exception through its `var` attribute. `var` is removed if no exception has occurred.

When this specification requires an action to throw an exception, this exception must be an instance of `javax.servlet.jsp.JspException` or a subclass. If an action catches any exceptions that occur in its body, its tag handler must provide the caught exception as the root cause of the `JspException` it re-throws.

Also, by default, JSTL actions do not catch or otherwise handle exceptions that occur during evaluation of their body content. If they do, it is documented in their "Null & Error Handling" or "Description" section.

2.8   Configuration Data

Context initialization parameters (see Servlet specification) are useful to configure the behavior of actions. For example, it is possible in JSTL to define the resource bundle used by I18N actions via the deployment descriptor (web.xml) as follows:

```
<web-app>
  ...
  <context-param>
    <param-name>javax.servlet.jsp.jstl.fmt.localizationContext</
param-name>
    <param-value>com.acme.MyResources</param-value>
  </context-param>
  ...
</web-app>
```

In many cases, it is also useful to allow configuration data to be overridden dynamically for a particular JSP scope (page, request, session, application) via a scoped variable. JSTL refers to scoped variables used for that purpose as *configuration variables.*

According to the JSP specification (JSP.2.8.2), a scoped variable name should refer to a unique object at all points in the execution. This means that all the different scopes (page, request, session, and application) that exist within a `PageContext` really should behave as a single name space; setting a scoped variable in any one scope overrides it in any of the other scopes.

Given this constraint imposed by the JSP specification, and in order to allow a configuration variable to be set for a particular scope without affecting its settings in any of the other scopes, JSTL provides the `Config` class (see Section 2.8.1). The `Config` class transparently manipulates the name of configuration variables so they behave as if scopes had their own private name space. Details on the name manipulations involved are voluntarily left unspecified and are handled transparently by the `Config` class. This ensures flexibility should the "scope name space" issue be addressed in the future by the JSP specification.

When setting configuration data via the deployment descriptor, the name associated with the context initialization parameter (e.g. `javax.servlet.jsp.jstl.fmt.localizationContext`) must be used and only `String` values may be specified. Configuration data that can be set both through a context initialization parameter and configuration variables is referred to as a *configuration setting* in this specification.

As mentioned above, application developers may access configuration data through class Config. As a convenience, constant String values have been defined in the Config class for each configuration setting supported by JSTL. The values of these constants are the names of the context intialization parameters.

Each configuration variable clearly specifies the Java data type(s) it supports. If the type of the object used as the value of a configuration variable does not match one of those supported by the configuration variable, conversion is performed according to the conversion rules defined in the expression language (see Section A.7). Setting a configuration variable is therefore exactly the same as setting an attribute value of an action using the EL. A failure of these conversion rules to determine an appropriate type coersion leads to a JspException at runtime.

2.8.1   Class Config

This class supports configuration data as specified in Section 2.8 of the JSTL specification.

Syntax

```
javax.servlet.jsp.jstl.core
public class Config
```

Constants

```
/**
  * I18N/Formatting actions related configuration data
  */
public static final String FMT_LOCALE =
                "javax.servlet.jsp.jstl.fmt.locale";
public static final String FMT_FALLBACK_LOCALE =
                "javax.servlet.jsp.jstl.fmt.fallbackLocale";
public static final String FMT_LOCALIZATION_CONTEXT =
                "javax.servlet.jsp.jstl.fmt.localizationContext";
public static final String FMT_TIME_ZONE =
                "javax.servlet.jsp.jstl.fmt.timeZone";



/**
  * SQL actions related configuration data.
  */
public static final String SQL_DATA_SOURCE =
                "javax.servlet.jsp.jstl.sql.dataSource";
public static final String SQL_MAX_ROWS =
```

```
                    "javax.servlet.jsp.jstl.sql.maxRows";
```

Methods

```
public static Object get(PageContext pageContext,
                          String name,
                          int scope);
```

Looks up a configuration variable in the given scope. The lookup of configuration variables is performed as if each scope had its own name space; i.e. the same configuration variable name in one scope does not replace one stored in a different scope.

Returns
: The `java.lang.Object` associated with the configuration variable. Null if it is not defined.

```
public static Object get(ServletRequest request,
                          String name);
```

Looks up a configuration variable in the "request" scope. The lookup of configuration variables is performed as if each scope had its own name space; i.e. the same configuration variable name in one scope does not replace one stored in a different scope.

Returns
: The `java.lang.Object` associated with the configuration variable. Null if it is not defined.

```
public static Object get(HttpSession session,
                          String name);
```

Looks up a configuration variable in the "session" scope. The lookup of configuration variables is performed as if each scope had its own name space; i.e. the same configuration variable name in one scope does not replace one stored in a different scope.

Returns
: The `java.lang.Object` associated with the configuration variable. Null if it is not defined.

```
public static Object get(ServletContext context,
                          String name);
```

Looks up a configuration variable in the "application" scope. The lookup of configuration variables is performed as if each scope had its own name space; i.e. the same configuration variable name in one scope does not replace one stored in a different scope.

Returns
: The `java.lang.Object` associated with the configuration variable. Null if it is not defined.

```
public static void set(PageContext pageContext,
                       String name,
                       Object var,
                       int scope);
```

Sets the value of a configuration variable in the given scope. Setting the value of a configuration variable is performed as if each scope had its own namespace; i.e. the same configuration variable name in one scope does not replace one stored in a different scope.

```
public static void set(ServletRequest request,
                       String name,
                       Object var);
```

Sets the value of a configuration variable in the "request" scope. Setting the value of a configuration variable is performed as if each scope had its own namespace; i.e. the same configuration variable name in one scope does not replace one stored in a different scope.

```
public static void set(HttpSession session,
                       String name,
                       Object var);
```

Sets the value of a configuration variable in the "session" scope. Setting the value of a configuration variable is performed as if each scope had its own namespace; i.e. the same configuration variable name in one scope does not replace one stored in a different scope.

```
public static void set(ServletContext context,
                       String name,
                       Object var);
```

Sets the value of a configuration variable in the "application" scope. Setting the value of a configuration variable is performed as if each scope had its own namespace; i.e. the same configuration variable name in one scope does not replace one stored in a different scope.

```
public static Object find(PageContext pageContext,
                          String name);
```

Finds the value associated with a specific configuration setting identified by its context initialization parameter name (argument `name`).

For each of the JSP scopes (page, request, session (if a session is present for the given `PageContext` object), and application), get the value of the configuration variable identified by `name` using method `get()`. Return as soon as a non-null value is found. If no value is found, get the value of the context initialization parameter identifed by `name`.

Returns

The `java.lang.Object` associated with the configuration setting identified by `name`. Null if it is not defined.

```
public static void remove(PageContext pageContext,
                          String name,
                          int scope);
```

Removes a configuration variable from the given scope. Removing a configuration variable is performed as if each scope had its own namespace; i.e. the same configuration variable name in one scope does not impact one stored in a different scope.

```
public static void remove(ServletRequest request,
                          String name);
```

Removes a configuration variable from the "request" scope. Removing a configuration variable is performed as if each scope had its own namespace; i.e. the same configuration variable name in one scope does not impact one stored in a different scope.

```
public static void remove(HttpSession session,
                          String name);
```

Removes a configuration variable from the "session" scope. Removing a configuration variable is performed as if each scope had its own namespace; i.e. the same configuration variable name in one scope does not impact one stored in a different scope.

```
public static void remove(ServletContext context,
                          String name);
```

Removes a configuration variable from the "application" scope. Removing a configuration variable is performed as if each scope had its own namespace; i.e. the same configuration variable name in one scope does not impact one stored in a different scope.

Expression Language Overview

JSTL introduces the notion of an expression language (EL) to make it easy for page authors to access and manipulate application data without having to master the complexity associated with programming languages such as Java and JavaScript.

As mentioned in Section 1.3, the specification of an expression language is ultimately the responsibility of the JSR-152 expert group (JSP version 1.3). The JSR-52 (JSTL) and JSR-152 expert groups have worked closely together on a specification of the expression language that meets the requirements identified in JSR-52. Given that JSTL 1.0 is finalized prior to JSP1.3, the goal has been to ensure that the current specification of the EL will be compatible with the one that will make it in the final draft of JSR-152. The plan is to then have a maintenance release of JSTL that will be based on the official expression language specified in the final draft of JSR-152.

In later releases (starting with JSTL 1.1), the EL will be formally defined within the JSP specification. For JSTL 1.0, the EL is defined in Appendix A.

This chapter provides a simple overview of the key features of the expression language, it is therefore non-normative. Please refer to Appendix A for the formal definition of the EL.

3.1   Expressions and Attribute Values

In JSTL, the EL is available only in attribute values[1]. In the sample code below, attributes test (in action <c:if>) and value (in action <c:out>) both have their values specified as EL expressions.

```
<c:if test="${book.price <= user.preferences.spendingLimit}">
   The book <c:out value="${book.title}"/> fits your budget!
</c:if>
```

The EL is invoked exclusively via the construct ${*expr*}.

It is also possible for an attribute to contain more than one EL expression, mixed with static text. For example, the following would display "Price of *productName* is *productPrice*" for a list of products.

```
<c:forEach var="product" items="${products}">
   <c:out value="Price of ${product.name} is ${product.price}"/>
</c:forEach>
```

3.2   Accessing Application Data

An identifier in the EL refers to the JSP scoped variable returned by a call to PageContext.findAttribute(*identifier*). This variable can therefore reside in any of the four JSP scopes: page, request, session, or application. A null value is returned if the variable does not exist in any of the scopes.

The EL also defines implicit objects to support easy access to application data that is of interest to a page author. Implicit objects pageScope, requestScope, sessionScope, and applicationScope provide access to the scoped variables in each one of these JSP scopes. It is also possible to access HTTP request parameters via the implicit objects param and paramValues. param is a Map object where param["foo"] returns the first string value associated with request parameter foo, while paramValues["foo"] returns an array of all string values associated with that request parameter.

---

1. When the EL is integrated within the JSP specification (JSP1.3), it will also be possible to use the EL within template text.

The code below displays all request parameters along with all their associated values.

```
<c:forEach var="aParam" items="${paramValues}">
    param: <c:out value="${aParam.key}"/>
    values:
    <c:forEach var="aValue" items="${aParam.value}">
        <c:out value="${aValue}"/>
    </c:forEach>
    <br>
</c:forEach>
```

Request headers are also accessible in a similar fashion via implicit objects `header` and `headerValues`. `initParam` gives access to context initialization parameters, while `cookie` exposes cookies received in the request.

Implicit object `pageContext` is also provided for advanced usage, giving access to all properties associated with the `PageContext` of a JSP page such as the `HttpServletRequest`, `ServletContext`, and `HttpSession` objects and their properties.

3.3   Nested Properties and Accessing Collections

The application data that a page author manipulates in a JSP page usually consists of objects that comply with the JavaBeans specification, or that represent collections such as lists, maps, or arrays.

The EL recognizes the importance of these data structures and provides two operators, "." and "[]", to make it easy to access the data encapsulated in these objects.

The "." operator can be used as a convenient shorthand for property access when the property name follows the conventions of Java identifiers. For example:

```
Dear <c:out value="${user.firstName}"/>
from <c:out value="${user.address.city}"/>,
thanks for visiting our website!
```

The "[]" operator allows for more generalized access, as shown below:

```
<%-- "productDir" is a Map object containing the description of
     products, "preferences" is a Map object containing the
     preferences of a user --%>
product:
<c:out value="${productDir[product.custId]}"/>
shipping preference:
<c:out value='${user.preferences["shipping"]}'/>
```

3.4   Operators

The operators supported in the EL handle the most common data manipulations. The standard relational, arithmetic, and logical operators are provided in the EL. A very useful "empty" operator is also provided.

The six standard relational operators are supported: == (or eq), != (or ne), < (or lt), > (or gt), <= (or le), >= (or ge). The second versions of the last 4 operators are made available to avoid having to use entity references in XML syntax.

Arithmetic operators consist of addition (+), substraction (–), multiplication (*), division (/ or div), and remainder/modulo (% or mod).

Logical operators consist of && (or and), || (or or), and ! (or not).

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

```
<c:if test="${empty param.name}">
  Please specify your name.
</c:if>
```

3.5   Automatic Type Conversion

The application data a page author has access to may not always exactly match the type expected by the attribute of an action or the type expected for an EL operator. The EL supports an exhaustive set of rules to coerce the type of the resulting value to the expected type.

For example, if request attributes `beginValue` and `endValue` are `Integer` objects, they will automatically be coerced to `int`s when used with the <c:forEach> action.

```
<c:forEach  begin="${requestScope.beginValue}"
            end="${requestScope.endValue}">
   ...
</c:forEach>
```

In the example below, the parameter String value `param.start` is coerced to a number and is then added to 10 to yield an int value for attribute `begin`.

```
<c:forEach  items="${products}" begin="${param.start + 10}">
   ...
</c:forEach>
```

3.6   Default Values

JSP pages are mostly used in presentation. Experience suggests that it is important to be able to provide as good a presentation as possible, even when simple errors occur in the page. To satisfy this requirement, the EL provides default values rather than errors when failure to evaluate an expression is deemed "recoverable". Default values are type-correct values that allow a page to easily recover from these error conditions.

In the following example, the expression `"${user.address.city}"` evaluates to `null` rather than throwing a `NullPointerException` if there is no address associated with the `user` object. This way, a sensible default value can be displayed without having to worry about exceptions being thrown by the JSP page.

```
City: <c:out value="${user.address.city}" default="N/A"/>
```

In the following example, the addition operator considers the value of `param.start` to be 0 if it is not defined, therefore evaluating the expression to 10.

```
<c:forEach  items="${products}" begin="${param.start + 10}">
   ...
</c:forEach>
```

General-Purpose Actions
core tag library

This chapter introduces general purpose actions to support the manipulation of scoped variables as well as to handle error conditions.

4.1   Overview

Because the expression language is not yet part of the JSP specification, it is essential to provide a capability similar to JSP expressions (<%= *scripting-language-expression* %>) to easily display the value of EL expressions. The <c:out> action provides this capability. For example:

```
You have <c:out value="${sessionScope.user.itemCount}"/> items.
```

By default, <c:out> converts the characters <, >, ', ", & to their corresponding character entity codes (e.g. < is converted to <). If these characters are not converted, the page may not be rendered properly by the browser, and it could also open the door for cross-site scripting attacks (e.g. someone could post JavaScript code for closing the window to an online discussion forum). The conversion may be bypassed by specifying false to the escapeXml attribute.

The <c:out> action also supports the notion of default values for cases where the value of an EL expression is null. In the example below, the value "unknown" will be displayed if the property city is not accessible.

```
<c:out value="${customer.address.city}" default="unknown"/>
```

The action <c:set> is used to set the value of a JSP scoped attribute as follows:

```
<c:set var="foo" value="elexprvalue"/>
```

It is also possible to set the value of a scoped variable (JSP scoped attribute) from the body of the <c:set> action. This solves the problem associated with not being able to set an attribute value from another action. In the past, a tag developer would often implement extra "attributes as tags" so the value of these attributes could be set from other actions.

For example, the action <acme:att1> was created only to support setting the value of att1 of the parent tag <acme:atag> from other actions .

```
<acme:atag>
    <acme:att1>
        <acme:foo>mumbojumbo</acme:foo>
    </acme:att1>
</acme:atag>
```

With the <c:set> tag, this can be handled without requiring the extra <acme:att1> tag.

```
<c:set var="att1">
    <acme:foo>mumbojumbo</acme:foo>
</c:set>
<acme:atag att1="${att1}"/>
```

In the preceding example, the <c:set> action sets the value of the att1 scoped variable to the output of the <acme:foo> action. <c:set> – like all JSTL actions that create scoped attributes – creates scoped attributes in "page" scope by default.

<c:set> may also be used to set the property of a JavaBeans object, or add or set a specific element in a java.util.Map object. For example:.

```
<!-- set property in JavaBeans object -->
<c:set target="${cust.address}" property="city" value="${city}"/>

<!-- set/add element in Map object -->
<c:set target="${preferences}" property="color"
        value="${param.color}"/>
```

Action <c:remove> is the natural companion to <c:set>, allowing the explicit removal of scoped variables. For example:

```
<c:remove var="cachedResult" scope="application"/>
```

Finally, the <c:catch> action provides a complement to the JSP error page mechanism. It is meant to allow page authors to recover gracefully from error conditions that they can control. For example:

```
<c:catch var="exception">
  <!-- Execution we can recover from if exception occurs -->
  ...
</c:catch>
<c:if test="${exception != null}">
  Sorry. Processing could not be performed because...
</c:if>
```

4.2   \<c:out>

Evaluates an expression and outputs the result of the evaluation to the current `JspWriter` object.

Syntax

Without a body
```
<c:out value="value" [escapeXml="{true|false}"]
      [default="defaultValue"] />
```

With a body
```
<c:out value="value" [escapeXml="{true|false}"]>
   default value
</c:out>
```

Body Content

JSP. The JSP container processes the body content, then the action trims it and processes it further.

Attributes

Name	Dyn	Type	Description
value	true	Object	Expression to be evaluated.
escapeXml	true	boolean	Deterrmines whether characters <,>,&,',” in the resulting string should be converted to their corresponding character entity codes. Default value is true.
default	true	Object	Default value if the resulting value is null.

Null & Error Handling

- If `value` is null, the default value takes over. If no default value is specified, it itself defaults to an empty string.

Description

The expression to be evaluated is specified via the `value` attribute. The result of

the evaluation is coerced to a `String` and is subsequently emitted into the current `JspWriter` object.

If `escapeXml` is true, the following character conversions are applied:

Character	Character Entity Code
<	<
>	>
&	&
'	'
"	"

The default value can be specified either via the `default` attribute (using the syntax without a body), or within the body of the tag (using the syntax with a body). It defaults to an empty string.

4.3   <c:set>

Sets the value of a scoped variable or a property of a target object.

Syntax

Syntax 1: Set the value of a scoped variable using attribute value
```
<c:set value="value"
       var="varName" [scope="{page|request|session|application}"]/>
```

Syntax 2: Set the value of a scoped variable using body content
```
<c:set var="varName" [scope="{page|request|session|application}"]>
    body content
</c:set>
```

Syntax 3: Set a property of a target object using attribute value
```
<c:set value="value"
       target="target" property="propertyName"/>
```

Syntax 4: Set a property of a target object using body content
```
<c:set target="target" property="propertyName">
    body content
</c:set>
```

Body Content

JSP. The JSP container processes the body content, then the action trims it and processes it further.

Attributes

Name	Dyn	Type	Description
value	true	Object	Expression to be evaluated.
var	false	String	Name of the exported scoped variable to hold the value specified in the action. The type of the scoped variable is whatever type the value expression evaluates to.
scope	false	String	Scope for var.
target	true	Object	Target object whose property will be set. Must evaluate to a JavaBeans object with setter property property, or to a java.util.Map object.
property	true	String	Name of the property to be set in the target object.

Null & Error Handling

- Syntax 3 and 4: Throw an exception under any of the following conditions:
  - `target` evaluates to null
  - `target` is not a `java.util.Map` object and is not a JavaBeans object that supports setting property `property`.
- If `value` is null
  - Syntax 1: the scoped variable defined by `var` and `scope` is removed.
    - If attribute `scope` is specified, the scoped variable is removed according to the semantics of `PageContext.removeAttribute(varName, scope)`.
    - Otherwise, the scoped variable is removed according to the semantics of `PageContext.removeAttribute(varName)`.
  - Syntax 3:
    - if `target` is a `Map`, remove the entry with the key identified by `property`.
    - if `target` is a JavaBean component, set the property to null.

Description

Syntax 1 and 2 set the value of a the scoped variable identified by `var` and `scope`.

Syntax 3 and 4:
- If the target expression evaluates to a `java.util.Map` object, set the value of the element associated with the key identified by `property`. If the element does not exist, add it to the `Map` object.
- Otherwise, set the value of the property `property` of the JavaBeans object `target`. If the type of the value to be set does not match the type of the bean property, conversion is performed according to the conversion rules defined in the expression language (see Section A.7). With the exception of a null value, setting a bean property with <c:set> is therefore exactly the same as setting an attribute value of an action using the EL. A failure of these conversion rules to determine an appropriate type coersion leads to a `JspException` at runtime.

4.4 <c:remove>

Removes a scoped variable.

Syntax

```
<c:remove var="varName"
         [scope="{page|request|session|application}"]/>
```

Attributes

Name	Dynamic	Type	Description
var	false	String	Name of the scoped variable to be removed.
scope	false	String	Scope for var.

Description

The <c:remove> action removes a scoped variable.

If attribute scope is not specified, the scoped variable is removed according to the semantics of PageContext.removeAttribute(*varName*). If attribute scope is specified, the scoped variable is removed according to the semantics of PageContext.removeAttribute(*varName*, *scope*).

4.5   <c:catch>

Catches a `java.lang.Throwable` thrown by any of its nested actions.

Syntax

```
<c:catch [var="varName"]>
    nested actions
</c:catch>
```

Body Content

JSP. The body content is processed by the JSP container and the result is written to
the current `JspWriter`.

Attributes

Name	Dynamic	Type	Description
var	false	String	Name of the exported scoped variable for the exception thrown from a nested action. The type of the scoped variable is the type of the exception thrown.

Description

The <c:catch> action allows page authors to handle errors from any action in a
uniform fashion, and allows for error handling for multiple actions at once.

<c:catch> provides page authors with granular error handling: Actions that are of
central importance to a page should not be encapsulated in a <c:catch>, so their
exceptions will propagate to an error page, whereas actions with secondary
importance to the page should be wrapped in a <c:catch>, so they never cause the
error page mechanism to be invoked.

The exception thrown is stored in the scoped variable identified by `var`, which
always has page scope. If no exception occurred, the scoped variable identified by
`var` is removed if it existed.

If `var` is missing, the exception is simply caught and not saved.

Conditional Actions
core tag library

The output of a JSP page is often conditional on the value of dynamic application data. A simple scriptlet with an `if` statement can be used in such situations, but this forces a page author to use a scripting language whose syntax may be troublesome (e.g. one may forget the curly braces).

The JSTL conditional actions make it easy to do conditional processing in a JSP page.

5.1 Overview

The JSTL conditional actions are designed to support the two most common usage patterns associated with conditional processing: simple conditional execution and mutually exclusive conditional execution.

A *simple* conditional execution action evaluates its body content only if the test condition associated with it is true. In the following example, a special greeting is displayed only if this is a user's first visit to the site:

```
<c:if test="${user.visitCount == 1}">
   This is your first visit. Welcome to the site!
</c:if>
```

With *mutually exclusive* conditional execution, only one among a number of possible alternative actions gets its body content evaluated.

For example, the following sample code shows how the text rendered depends on a user's membership category.

```
<c:choose>
   <c:when test="${user.category == 'trial'}">
      ...
   </c:when>
   <c:when test="${user.category == 'member'}">
      ...
   </c:when>
   <c:when test="${user.category == 'vip'}">
      ...
   </c:when>
   <c:otherwise>
      ...
   </c:otherwise>
</c:choose>
```

An if/then/else statement can be easily achieved as follows:

```
<c:choose>
   <c:when test="${count == 0}">
      No records matched your selection.
   </c:when>
   <c:otherwise>
      <c:out value="${count}"/> records matched your selection.
   </c:otherwise>
</c:choose>
```

5.2   <c:if>

Evaluates its body content if the expression specified with the `test` attribute is true.

Syntax

Syntax 1: Without body content

```
<c:if test="testCondition"
      var="varName" [scope="{page|request|session|application}"]/>
```

Syntax 2: With body content

```
<c:if test="testCondition"
      [var="varName"] [scope="{page|request|session|application}"]>
   body content
</c:if>
```

Body Content

JSP. If the test condition evaluates to true, the JSP container processes the body content and then writes it to the current `JspWriter`.

Attributes

Name	Dyn	Type	Description
test	true	boolean	The test condition that determines whether or not the body content should be processed.
var	false	String	Name of the exported scoped variable for the resulting value of the test condition. The type of the scoped variable is `Boolean`.
scope	false	String	Scope for `var`.

Constraints

- If `scope` is specified, `var` must also be specified.

Description

If the test condition evaluates to true, the body content is evaluated by the JSP container and the result is output to the current `JspWriter`.

5.3   <c:choose>

Provides the context for mutually exclusive conditional execution.

Syntax

```
<c:choose>
   body content (<when> and <otherwise> subtags)
</c:choose>
```

Body Content

JSP. The body content is processed by the JSP container (at most one of the nested actions will be processed) and written to the current `JspWriter`.

Attributes

None.

Constraints

- The body of the <c:choose> action can only contain:
    - White spaces

      May appear anywhere around the <c:when> and <c:otherwise> subtags.
    - 1 or more <c:when> actions

      Must all appear before <c:otherwise>
    - 0 or 1 <c:otherwise> action

      Must be the last action nested within <c:choose>

Description

The <c:choose> action processes the body of the first <c:when> action whose test condition evaluates to true. If none of the test conditions of nested <c:when> actions evaluates to true, then the body of an <c:otherwise> action is processed, if present.

5.4   `<c:when>`

Represents an alternative within a <c:choose> action.

Syntax

```
<c:when test="testCondition">
   body content
</c:when>
```

Body Content

JSP. If this is the first <c:when> action to evaluate to true within <c:choose>, the JSP container processes the body content and then writes it to the current `JspWriter`.

Attributes

Name	Dynamic	Type	Description
`test`	true	`boolean`	The test condition that determines whether or not the body content should be processed.

Constraints

- Must have <c:choose> as an immediate parent.
- Must appear before an <c:otherwise> action that has the same parent.

Description

Within a <c:choose> action, the body content of the first <c:when> action whose test condition evaluates to true is evaluated by the JSP container, and the result is output to the current `JspWriter`.

5.5 <c:otherwise>

Represents the last alternative within a <c:choose> action.

Syntax

```
<c:otherwise>
    conditional block
</c:otherwise>
```

Body Content

JSP. If no <c:when> action nested within <c:choose> evaluates to true, the JSP container processes the body content and then writes it to the current `JspWriter`.

Attributes

None.

Constraints

- Must have <c:choose> as an immediate parent.
- Must be the last nested action within <c:choose>.

Description

Within a <c:choose> action, if none of the nested <c:when> test conditions evaluates to true, then the body content of the <c:otherwise> action is evaluated by the JSP container, and the result is output to the current `JspWriter`.

5.6 Custom Logic Actions

It is important to note that the <c:if> and <c:when> actions have different semantics. A <c:if> action will always process its body content if its test condition evaluates to true. A <c:when> action will process its body content if it is the *first* one in a series of <c:when> actions whose test condition evaluates to true.

These semantic differences are enforced by the fact that only <c:when> actions can be used within the context of a mutually exclusive conditional execution (<c:choose> action). This clean separation of behavior also impacts the way custom logic actions (i.e. actions who render their bodies depending on the result of a test condition) should be designed. Ideally, the result associated with the evaluation of a custom logic action should be usable both in the context of a simple conditional execution, as well as in a mutually exclusive conditional execution.

The proper way to enable this is by simply having the custom logic action export the result of the test condition as a scoped variable. This boolean result can then be used as the test condition of a <c:when> action.

In the example below, the fictitious custom action <acme:fullMoon> tells whether or not a page is accessed during a full moon. The behavior of an if/then/else statement is made possible by having the result of the <acme:fullMoon> action exposed as a boolean scoped variable that is then used as the test condition in the <c:when> action.

```
<acme:fullMoon var="isFullMoon"/>
<c:choose>
  <c:when test="${isFullMoon}">
    ...
  </c:when>
  <c:otherwise>
    ...
  </c:otherwise>
</c:choose>
```

5.7   Class ConditionalTagSupport

Abstract class that facilitates implementation of conditional actions where the boolean result is exposed as a JSP scoped variable. The boolean result may then be used as the test condition in a <c:when> action.

This base class provides support for:

- Conditional processing of the action's body based on the returned value of the abstract method `condition()`.

- Storing the result of `condition()` as a `Boolean` object into a JSP scoped variable identified by attributes `var` and `scope`.

Syntax

```
javax.servlet.jsp.jstl.core
public abstract class ConditionalTagSupport
    extends javax.servlet.jsp.tagext.TagSupport
```

Constructor

```
public ConditionalTagSupport()
```

Base constructor to initialize local state. As with `TagSupport`, subclasses should not implement constructors with arguments, and no-argument constructors implemented by subclasses must call the superclass constructor.

Methods

```
protected abstract boolean condition() throws JspTagException
```

Subclasses implement this method to compute the boolean result of the conditional action. This method is invoked once per invocation (by `doStartTag()`).

Returns
  A boolean representing the condition that a particular subclass uses to drive its conditional logic.

```
public void setVar(String var)
```

Sets the attribute `var`.

```
public void setScope(String scope)
```

Sets the attribute `scope`.

Iterator Actions
core tag library

Iterating over a collection of objects is a common occurrence in a JSP page. Just as with conditional processing, a simple scriptlet can be used in such situations. However, this once again forces a page author to be knowledgeable in many aspects of the Java programming language (how to iterate on various collection types, having to cast the returned object into the proper type, proper use of the curly braces, etc.).

The JSTL iterator actions simplify iterating over a wide variety of collections of objects.

6.1   Overview

The <c:forEach> action repeats its nested body content over the collection of objects specified by the items attribute. For example, the JSP code below creates an HTML table with one column that shows the default display value of each item in the collection.

```
<table>
    <c:forEach var="customer" items="${customers}">
        <tr><td><c:out value="${customer}"/></td></tr>
    </c:forEach>
</table>
```

The <c:forEach> action has the following features:

- Supports all standard J2SE™ platform collection types.

  A page author therefore does not have to worry about the specific type of the collection of objects to iterate over (see Section 6.1.1).

- Exports an object that holds the current item of the iteration.

  Normally, each object exposed by <c:forEach> is an item of the underlying collection being iterated over. There are two exceptions to this to facilitate access to the information contained in arrays of primitive types, as well as in `Map` objects (see Section 6.1.2).

- Exports an object that holds information about the status of the iteration (see Section 6.1.3).

- Supports range attributes to iterate over a subset of the original collection (see Section 6.1.4).

- Exposes an interface as well as a base implementation class.

  Developers can easily implement collaborating subtags as well as their own iteration tags (see Section 6.1.5).

<c:forEach> is the base iteration action in JSTL. It handles the most common iteration cases conveniently. Other iteration actions are also provided in the tag library to support specific, specialized functionality not handled by <c:forEach> (e.g. <c:forTokens> (Section 6.3) and <x:forEach> (Section 12.6)). Developers can also easily extend the behavior of this base iteration action to customize it according to an application's specific needs.

6.1.1   Collections of objects to iterate over

A large number of *collection* types are supported by <c:forEach>, including all implementations of `java.util.Collection` (includes `List, LinkedList, ArrayList, Vector, Stack, Set`), and `java.util.Map` (includes `HashMap, Hashtable, Properties, Provider, Attributes`).

Arrays of objects as well as arrays of primitive types (e.g. `int`) are also supported. For arrays of primitive types, the current item for the iteration is automatically wrapped with its standard wrapper class (e.g. `Integer` for `int`, `Float` for `float`, etc.).

Implementations of `java.util.Iterator` and `java.util.Enumeration` are supported as well but these must be used with caution. `Iterator` and `Enumeration` objects are not resettable so they should not be used within more than one iteration tag.

Finally, `java.lang.String` objects can be iterated over if the string represents a list of comma separated values (e.g. "Monday,Tuesday,Wednesday,Thursday,Friday").

Absent from the list of supported types is `java.sql.ResultSet` (which includes `javax.sql.RowSet`). The reason for this is that the SQL actions described in Section 10.1 use the `javax.servlet.jsp.jstl.sql.Result` interface to access the data returned from an SQL query. Class

`javax.servlet.jsp.jstl.sql.ResultSupport` (see Section 10.11) allows business logic developers to easily convert a `ResultSet` object into a `javax.servlet.jsp.jstl.sql.Result` object, making life much easier for a page author that needs to manipulate the data returned from a SQL query.

6.1.2   Map

If the `items` attribute is of type `java.util.Map`, then the current item will be of type `java.util.Map.Entry`, which has the following two properties:

- `key` - the key under which this item is stored in the underlying `Map`
- `value` - the value that corresponds to this key

The following example uses <c:forEach> to iterate over the values of a `Hashtable`:

```
<c:forEach var="entry" items="${myHashtable}">
    Next element is <c:out value="${entry.value}"/>
</c:forEach>
```

6.1.3   Iteration Status

<c:forEach> also exposes information relative to the iteration taking place. The example below creates an HTML table with the first column containing the position of the item in the collection, and the second containing the name of the product.

```
<table>
    <c:forEach var="product" items="${products}"
               varStatus="status">
        <tr>
            <td><c:out value="${status.count}"/></td>
            <td><c:out value="${product.name}"/></td>
        </tr>
    </c:forEach>
</table>
```

See Section 6.4 for details on the `LoopTagStatus` interface exposed by the `varStatus` attribute.

6.1.4 Range Attributes

A set of range attributes is available to iterate over a subset of the collection of items. The `begin` and `end` indices can be specified, along with a `step`. If the `items` attribute is not specified, then the value of the current item is set to the integer value of the current index. In this example, `i` would take values from 100 to 110 (inclusive).

```
<c:forEach var="i" begin="100" end="110">
    <c:out value="${i}"/>
</c:forEach>
```

6.1.5 Tag Collaboration

Custom actions give developers the power to provide added functionality to a JSP application without requiring the page author to use Java code. In this example, an item of the iteration is processed differently depending upon whether it is an odd or even element.

```
<c:forEach var="product" items="${products}" varStatus="status">
    <c:choose>
        <c:when test="${status.count % 2 == 0}">
            even item
        </c:when>
        <c:otherwise>
            odd item
        </c:otherwise>
    </c:choose>
</c:forEach>
```

If this type of processing is common, it could be worth providing custom actions that yield simpler code, as shown below.

```
<c:forEach var="product" items="${products}">
    <acme:even>
        even item
    </acme:even>
    <acme:odd>
        odd item
    </acme:odd>
</c:forEach>
```

In order to make this possible, custom actions like <acme:odd> and <acme:even> leverage the fact that <c:forEach> supports implicit collaboration via the interface `LoopTag` (see Section 6.5).

The fact that <c:forEach> exposes an interface also means that other actions with iterative behavior can be developed using the same interface and will collaborate in the same manner with nested tags. Class `LoopTagSupport` (see Section 6.6) provides a solid base for doing this.

6.2   \<c:forEach>

Repeats its nested body content over a collection of objects, or repeats it a fixed number of times.

Syntax

Syntax 1: Iterate over a collection of objects
```
<c:forEach[var="varName"] items="collection"
          [varStatus="varStatusName"]
   [begin="begin"] [end="end"] [step="step"]>

     body content

</c:forEach>
```

Syntax 2: Iterate a fixed number of times
```
<c:forEach [var="varName"]
        [varStatus="varStatusName"]
        begin="begin" end="end" [step="step"]>

     body content

</c:forEach>
```

Body Content

JSP. As long as there are items to iterate over, the body content is processed by the JSP container and written to the current `JspWriter`.

Attributes

Name	Dyn	Type	Description
var	false	String	Name of the exported scoped variable for the current item of the iteration. This scoped variable has nested visibility. Its type depends on the object of the underlying collection.
items	true	Any of the supported types described in Section "Description" below.	Collection of items to iterate over.
varStatus	false	String	Name of the exported scoped variable for the status of the iteration. Object exported is of type `javax.servlet.jsp.jstl.core.LoopTagStatus`. This scoped variable has nested visibility.
begin	true	int	<u>If `items` specified:</u> Iteration begins at the item located at the specified index. First item of the collection has index 0. <u>If `items` not specified:</u> Iteration begins with index set at the value specified.
end	true	int	<u>If `items` specified:</u> Iteration ends at the item located at the specified index (inclusive). <u>If `items` not specified:</u> Iteration ends when index reaches the value specified.
step	true	int	Iteration will only process every `step` items of the collection, starting with the first one.

Constraints

- If specified, `begin` must be >= 0.
- If specified, `end` must be >= `begin`
- If specified, `step` must be >= 1

Null & Error Handling

- If `items` is null, it is treated as an empty collection, i.e., no iteration is performed.

Description

If `begin` is greater than or equal to the size of `items`, no iteration is performed.

Collections Supported & Current Item

The data types listed below must be supported for `items`. With syntax 1, each object exposed via the `var` attribute is of the type of the object in the underlying collection, except for arrays of primitive types and maps (see below). With syntax 2, the object exported is of type `Integer`.

■ Arrays

This includes arrays of objects as well as arrays of primitive types. For arrays of primitive types, the current item for the iteration is automatically wrapped with its standard wrapper class (e.g. `Integer` for `int`, `Float` for `float`, etc.)

Elements are processed in their indexing order.

■ Implementation of `java.util.Collection`.

An `Iterator` object is obtained from the collection via the `iterator()` method, and the items of the collection are processed in the order returned by that `Iterator` object.

■ Implementation of `java.util.Iterator`.

Items of the collection are processed in the order returned by the `Iterator` object.

■ Implementation of `java.util.Enumeration`.

Items of the collection are processed in the order returned by the `Enumeration` object.

■ Implementation of `java.util.Map`

The object exposed via the `var` attribute is of type `Map.Entry`.

A `Set` view of the mappings is obtained from the `Map` via the `entrySet()` method, from which an `Iterator` object is obtained via the `iterator()` method. The items of the collection are processed in the order returned by that `Iterator` object.

■ `String`

The string represents a list of comma separated values, where the comma character is the token delimiter. Tokens are processed in their sequential order in the string.

6.3 <c:forTokens>

Iterates over tokens, separated by the supplied delimiters.

Syntax

```
<c:forTokens items="stringOfTokens" delims="delimiters"
      [var="varName"]
      [varStatus="varStatusName"]
      [begin="begin"] [end="end"] [step="step"]>
   body content
</c:forEach>
```

Body Content

JSP. As long as there are items to iterate over, the body content is processed by the JSP container and written to the current `JspWriter`.

Attributes

Name	Dynamic	Type	Description
var	false	String	Name of the exported scoped variable for the current item of the iteration. This scoped variable has nested visibility.
items	true	String	String of tokens to iterate over.
delims	true	String	The set of delimiters (the characters that separate the tokens in the string).
varStatus	false	String	Name of the exported scoped variable for the status of the iteration. Object exported is of type `javax.servlet.jsp.jstl.core.LoopTagStatus`. This scoped variable has nested visibility.
begin	true	int	Iteration begins at the token located at the specified index. First token has index 0.
end	true	int	Iteration ends at the token located at the specified index (inclusive).
step	true	int	Iteration will only process every `step` tokens of the string, starting with the first one.

Constraints

- If specified, `begin` must be >= 0.
- If specified, `end` must be >= `begin`
- If specified, `step` must be >= 1

Null & Error Handling

- If `items` is null, it is treated as an empty collection, i.e., no iteration is performed.
- If `delims` is null, `items` is treated as a single monolithic token. Thus, when `delims` is null, <c:forTokens> iterates exactly zero (if `items` is also null) or one time.

Description

The tokens of the string are retrieved using an instance of `java.util.StringTokenizer` with arguments `items` (the string to be tokenized) and `delims` (the delimiters).

Delimiter characters separate tokens. A token is a maximal sequence of consecutive characters that are not delimiters.

6.4 Interface LoopTagStatus

Exposes the current status of an iteration.

Syntax

```
javax.servlet.jsp.jstl.core
public interface LoopTagStatus
```

Methods

```
public java.lang.Object getCurrent()
```

Retrieves the current item in the iteration. This function is idempotent, i.e. calling `getCurrent()` repeatedly returns the same `Object` until the iteration is advanced (specifically, calling `getCurrent()` does not advance the iteration.)

Returns

The current item.

```
public int getIndex()
```

Retrieves the index of the current round of the iteration. If the iteration is being performed over a subset of an underlying array, `java.lang.Collection`, or other type, the index returned is absolute with respect to the underlying collection. Indices are 0-based.

Returns

The 0-based index of the current round of the iteration

```
public int getCount()
```

Retrieves the count of the current round of the iteration. The count is a relative, 1-based sequence number identifying the current round of iteration (in context with all rounds the current iteration will perform). As an example, an iteration with begin = 5, end = 15, and step = 5 produces the counts 1, 2, and 3 in that order.

Returns

The 1-based count of the current round of the iteration

```
public boolean isFirst()
```

Returns information about whether the current round of the iteration is the first one. This current round may be the first even when `getIndex()` != 0, for index refers to the absolute index of the current item in the context of its underlying collection. It is always the case that a true result from `isFirst()` implies `getCount()` == 1.

Returns

True if the current round is the first in the iteration, false otherwise.

```
public boolean isLast()
```

Returns information about whether the current round of the iteration is the last one. As with isFirst(), subsetting is taken into account. isLast() doesn't necessarily refer to the status of the underlying iterator; it refers to whether or not the current round will be the final round of iteration for the tag associated with this LoopTagStatus.

<u>Returns:</u>

True if the current round is the last in the iteration, false otherwise.

```
public Integer getBegin()
```

Returns the value of the begin attribute for the associated tag.

<u>Returns</u>

The begin value for the associated tag, or null if this attribute was not specified.

```
public Integer getEnd()
```

Returns the value of the end attribute for the associated tag.

<u>Returns</u>

The end value for the associated tag, or null if this attribute was not specified.

```
public Integer getStep()
```

Returns the value of the step attribute for the associated tag.

<u>Returns</u>

The step value for the associated tag, or null if this attribute was not specified.

6.5 Interface LoopTag

`LoopTag` establishes a mechanism for iteration tags to be recognized and for type-safe implicit collaboration with custom subtags.

Developers can also provide their own custom iteration tags by implementing the `LoopTag` interface. In most cases, it will not be necessary to implement this interface manually, because a base support class (`LoopTagSupport`) is provided to facilitate implementation.

Syntax

```
javax.servlet.jsp.jstl.core
public interface LoopTag
    extends javax.servlet.jsp.tagext.Tag
```

Methods

`public java.lang.Object` **`getCurrent`**`()`

Retrieves the current item in the iteration. Behaves idempotently; calling `getCurrent()` repeatedly returns the same Object until the iteration is advanced (specifically, calling `getCurrent()` does not advance the iteration.)

Returns
  The current item.

`public LoopTagStatus` **`getLoopStatus`**`()`

Retrieves a status object to provide information about the current round of the iteration.

Returns
  The `LoopTagStatus` for the current `LoopTag`.

6.6   Class LoopTagSupport

JSTL allows developers to write custom iteration tags by implementing the `LoopTag` interface. (This is not to be confused with `javax.servlet.jsp.tagext.IterationTag` as defined in JSP 1.2.) `LoopTag` establishes a mechanism for iteration tags to be recognized and for type-safe communication with custom subtags.

Since most iteration tags will behave identically with respect to actual iterative behavior, JSTL provides this base support class to facilitate implementation. Many iteration tags will extend this and merely implement the `hasNext()` and `next()` methods to provide contents for the handler to iterate over.

In particular, this base class provides support for:

- Iteration control, based on protected `prepare()`, `next()`, and `hasNext()` methods
- Subsetting (`begin`, `end`, `step` functionality, including validation of subset parameters for sensibility)
- Item retrieval (`getCurrent()`)
- Status retrieval (`LoopTagStatus`)
- Exposing attributes (set by `var` and `status` attributes)

In providing support for these tasks, `LoopTagSupport` contains certain control variables that act to modify the iteration. Accessors are provided for these control variables when the variables represent information needed or wanted at translation time (e.g., `var`, `status`). For other variables, accessors cannot be provided here since subclasses may differ on their implementations of how those accessors are received. For instance, one subclass might accept a `String` and convert it into an object of a specific type by using an expression evaluator; others might accept objects directly. Still others might not want to expose such information to outside control.

Syntax

```
javax.servlet.jsp.jstl.core
public abstract class LoopTagSupport
    implements javax.servlet.jsp.tagext.IterationTag,
               javax.servlet.jsp.tagext.TryCatchFinally
```

Fields

```
protected int begin
```
  Starting index (`begin` attribute).

```
protected int end
```

Ending index (`end` attribute). -1 internally indicates no end specified, although accessors for the core JSTL tags do not allow this value to be supplied directly by the user.

```
protected int step
```

Iteration step (`step` attribute).

```
protected java.lang.String itemId
```

Attribute-exposing control.

```
protected String statusId
```

Attribute-exposing control.

```
protected boolean beginSpecified
```

Flag indicating whether the `begin` attribute was specified.

```
protected boolean endSpecified
```

Flag indicating whether the `end` attribute was specified.

```
protected boolean stepSpecified
```

Flag indicating whether the `step` attribute was specified.

Constructor

```
public LoopTagSupport()
```

Constructs a new `LoopTagSupport`. As with `TagSupport`, subclasses should not implement constructors with arguments, and no-argument constructors implemented by subclasses must call the superclass constructor.

Methods

```
protected abstract Object next() throws
javax.servlet.jsp.JspTagException
```

Returns the next object over which the tag should iterate. This method must be provided by concrete subclasses of `LoopTagSupport` to inform the base logic about what objects it should iterate over.

It is expected that this method will generally be backed by an `Iterator`, but this will not always be the case. In particular, if retrieving the next object raises the possibility of an exception being thrown, this method allows that exception to propagate back to the JSP container as a `JspTagException`; a standalone `Iterator` would not be able to do this. (This explains why `LoopTagSupport` does not simply call for an `Iterator` from its subtags.)

<u>Returns</u>

The `java.lang.Object` to use in the next round of iteration.

Throws

NoSuchElementException - if `next()` is called but no new elements are available

`javax.servlet.jsp.JspTagException` - for other, unexpected exceptions

```
protected abstract boolean hasNext() throws
javax.servlet.jsp.JspTagException
```

Returns information concerning the availability of more items over which to iterate. This method must be provided by concrete subclasses of `LoopTagSupport` to assist the iterative logic provided by the supporting base class.

See `next()` for more information about the purpose and expectations behind this tag.

Returns

`true` if there is at least one more item to iterate over, `false` otherwise.

```
protected abstract void prepare() throws
javax.servlet.jsp.JspTagException
```

Allows subclasses to prepare for calls to `hasNext()` and `next()` (e.g. for logic to reset an iteration). Subclasses can assume that `prepare()` will be called once for each invocation of `doStartTag()` in the parent. Base class implementation of this method is empty.

```
public Object getCurrent() throws javax.servlet.jsp.JspTagException
```

Returns

Current item of the iteration.

```
public LoopTagStatus getLoopStatus()
```

Returns

`LoopTagStatus` object associated with the iteration.

```
public void setVar(String varName)
```

Sets the attribute `var`.

```
public void setVarStatus(String statusName)
```

Sets the attribute `varStatus`.

```
protected void validateBegin()
    throws javax.servlet.jsp.JspTagException
```

Validates the value of the `begin` property and throws an exception if it is invalid.

```
protected void validateEnd()
    throws javax.servlet.jsp.JspTagException
```

Validates the value of the `end` property and throws an exception if it is invalid.

```
protected void validateStep()
   throws javax.servlet.jsp.JspTagException
```
Validates the value of the step property and throws an exception if it is invalid.

URL Related Actions

core tag library

Linking, importing, and redirecting to URL resources are features often needed in JSP pages. Since dealing with URLs can often be tricky, JSTL offers a comprehensive suite of URL-related actions to simplify these tasks.

7.1 Hypertext Links

By using the HTML <A> element, a page author can set a hypertext link as follows:

```
<a href="/register.jsp">Register</a>
```

If the link refers to a local resource and session tracking is enabled, it is necessary to *rewrite* the URL so session tracking can be used as a fallback, should cookies be disabled at the client.

Morevoer, if query string parameters are added to the URL, it is important that they be properly URL encoded. *URL encoding* refers to the process of encoding special characters in a string, according to the rules defined in RFC 2396. For example, a space must be encoded in a URL string as a '+':

```
http://acme.com/app/choose?country=Dominican+Republic
```

As shown in the following example, the combination of the <c:url> and <c:param> actions takes care of all issues related to URL rewriting and encoding: <c:url> rewrites a URL if necessary, and <c:param> transparently encodes query string parameters (both name and value).

```
<c:url value="http://acme.com/exec/register" var="myUrl">
    <c:param name="name" value="${param.name}"/>
    <c:param name="country" value="${param.country}"/>
</c:url>
<a href='<c:out value="${myUrl}"/>'>Register</a>
```

Another important feature of <c:url> is that it transparently prepends the context path to context-relative URLs. Assuming a context path of "/foo", the following example

```
<c:url value="/ads/logo.html"/>
```

yields the URL /foo/ads/logo.html.

_____

7.2   Importing Resources

There is a wide variety of resources that a page author might be interested in including and/or processing within a JSP page. For instance, the example below shows how the content of the README file at the FTP site of acme.com could be included within the page.

```
<c:import url="ftp://ftp.acme.com/README"/>
```

In the JSP specification, a <jsp:include> action provides for the inclusion of static and dynamic resources located in the same context as the current page. This is a very convenient feature that is widely used by page authors.

However, <jsp:include> falls short in flexibility when page authors need to get access to resources that reside outside of the web application. In many situations, page authors have the need to import the content of Internet resources specified via an absolute URL. Moreover, as sites grow in size, they may have to be implemented as a set of web applications where importing resources across web applications is a requirement.

<jsp:include> also falls short in efficiency when the content of the imported resource is used as the source for a companion process/transformation action, because unnecessary buffering occurs. In the example below, the <acme:transform> action uses the content of the included resource as the input of its transformation. <jsp:include> reads the content of the response, writes it to the body content of the enclosing <acme:transform>, which then re-reads the exact same content. It would be more efficient if <acme:transform> could access the input source directly and avoid the buffering involved in the body content of <acme:transform>.

```
<acme:transform>
    <jsp:include page="/exec/employeesList"/>
</acme:transform>
```

The main motivation behind <c:import> is to address these shortcomings by providing a simple, straightforward mechanism to access resources that can be specified via a URL. If accessing a resource requires specifying more arguments, then a protocol specific action (e.g. an <http> action) should be used for that purpose. JSTL 1.0 does not currently address these protocol-specific elements but may do so in future releases.

7.2.1   URL

The url attribute is used to specify the URL of the resource to import. It can either be an absolute URL (i.e. one that starts with a protocol followed by a colon), a relative URL used to access a resource within the same context, or a relative URL used to access a resource within a foreign context. The three different types of URL are shown in the sample code below.

```
<%-- import a resource with an absolute URL --%>
<c:import url="http://acme.com/exec/customers?country=Japan/>

<%-- import a resource with a relative URL - same context --%>
<c:import url="/copyright.html"/>

<%-- import a resource with a relative URL - foreign context --%>
<c:import url="/logo.html" context="/master"/>
```

7.2.2   Exporting an object: String or Reader

By default, the content of an imported resource is included inline into the JSP page.

It is also possible to make the content of the resource available in two different ways: as a `String` object (attribute `var`), or as a `Reader` object (attribute `varReader`). Process or Transform tags can then access the resource's content through that exported object as shown in the following example.

```
<%-- Export the content of the URL resource as a String --%>
<c:import url="http://acme.com/exec/customers?country=USA"
          var="customers"/>
<acme:notify in="${customers}"/>

<%-- Export the content of the URL resource as a Reader --%>
<c:import url="http://acme.com/exec/customers?country=USA"
          varReader="customers">
   <acme:notify in="${customers}"/>
</c:import>
```

Exporting the resource as a `String` object caches its content and makes it reusable.

If the imported content is large, some performance benefits may be achieved by exporting it as a `Reader` object since the content can be accessed directly without any buffering. However, the performance benefits are not guaranteed since the reader's support is implementation dependent. It is also important to note that the `varReader` scoped variable has nested visibility; it can only be accessed within the body content of <c:import>.

7.2.3   URL Encoding

Just as with <c:url>, <c:param> can be nested within <c:import> to encode query string parameters.

7.2.4   Networking Properties

If the web container executes behind a firewall, some absolute URL resources may be inaccessible when using <c:import>. To provide access to these resources, the JVM of the container should be started with the proper networking properties (e.g. `proxyHost`, `proxyPort`). More details can be found in the Java 2 SDK, Standard Edition Documentation (Networking Features — Networking Properties).

7.3   HTTP Redirect

<c:redirect> completes the arsenal of URL related actions to support an HTTP redirect to a specific URL. For example:

```
<c:redirect url="http://acme.com/register"/>
```

7.4   \<c:import\>

Imports the content of a URL-based resource.

Syntax

Syntax 1: Resource content inlined or exported as a String object

```
<c:import url="url" [context="context"]
        [var="varName"] [scope="{page|request|session|application}"]
        [charEncoding="charEncoding"]>
    optional body content for <c:param> subtags
</c:import>
```

Syntax 2: Resource content exported as a Reader object

```
<c:import url="url" [context="context"]
        varReader="varReaderName"
        [charEncoding="charEncoding"]>
    body content where varReader is consumed by another action
</c:import>
```

Body Content

JSP. The body content is processed by the JSP container and the result is written to the current `JspWriter`.

Attributes

Name	Dynamic	Type	Description
url	true	String	The URL of the resource to import.
context	true	String	Name of the context when accessing a relative URL resource that belongs to a foreign context.
var	false	String	Name of the exported scoped variable for the resource's content. The type of the scoped variable is `String`.

Name	Dynamic	Type	Description
scope	false	String	Scope for `var`.
charEncoding	true	String	Character encoding of the content at the input resource.
varReader	false	String	Name of the exported scoped variable for the resource's content. The type of the scoped variable is `Reader`.

Null & Error Handling

- If `url` is null, empty, or invalid, a `JspException` is thrown.
- If `charEncoding` is null or empty, it is considered missing.
- For internal resources:
  a) If a `RequestDispatcher` cannot be found for the resource, throw a `JspException` with the resource path included in the message.
  b) Otherwise, if the `RequestDispatcher.include()` method throws an `IOException` or `RuntimeException`, throw a `JspException` with the caught exception as the root cause.
  c) Otherwise, if the `RequestDispatcher.include()` method throws a `ServletException`, look for a root cause.
    - If there's a root cause, throw a `JspException` with the root cause message included in the message and the original root cause as the `JspException` root cause.
    - Otherwise, same as b).
  d) Otherwise, if the resource invoked through `RequestDispatcher.include()` method sets a response status code other than 2xx (i.e. 200-299, the range of success codes in the HTTP response codes), throw a `JspException` with the path and status code in the message.
- For external resources
  - If the `URLConnection` class throws an `IOException` or a `RuntimeException`, throw a `JspException` with the message from the original exception included in the message and the original exception as the root cause.
  - For an `HttpURLConnection`, if the response status code is other than 2xx (i.e. 200-299, the range of success codes in the HTTP response codes), throw a `JspException` with the path and status code in the message.

Description

Using syntax 1, the content of the resource is by default written to the current `JspWriter`. If `var` is specified, the content of the resource is instead exposed as a `String` object.

Using syntax 2, the content of the resource is exported as a `Reader` object. The use of the `varReader` attribute comes with some restrictions.

It is the responsibility of the <c:import> tag handler to ensure that if it exports a `Reader`, this `Reader` is properly closed by the time the end of the page is reached[1]. Because of this requirement, JSTL 1.0 defines the exported `Reader` as having nested visibility: it may not currently be accessed after the end-tag for the <c:import> action[2]. Implementations that use JSP 1.2 tag-extension API will likely need to implement `TryCatchFinally` with their <c:import> tag handlers and close the exported Reader in `doFinally()`.

It is also illegal to use nested <c:param> tags with syntax 2. Since the exposed `Reader` must be immediately available to the action's body, the connection to the resource must be established within the start element of the action. It is therefore impossible for nested <c:param> actions to modify the URL of the resource to be accessed, thus their illegality with syntax 2. In such a situation, <c:url> may be used to build a URL with query string parameters. <c:import> will remove any session id information if necessary (see Section 7.5).

Character Encoding

<c:import> exposes a `String` or `Reader` object, both of which are sequences of text characters (as opposed to binary data). If the content of the input resource is retrieved in binary form, it is possible to specify its character encoding via the `charEncoding` attribute so the binary data can be properly converted to text characters. The values supported for `charEncoding` are the same as the ones supported by the constructor of the Java class `InputStreamReader`.

If the character encoding is not specified, the following rules apply:

- If the response has content encoding information (e.g. `URLConnection.getContentEncoding()` has a non null value), then the character encoding specified is used.
- Use ISO-8859-1 (this is the default value of charset for the `contentType` attribute of the JSP "page" directive).

Note that the `charEncoding` attribute should normally only be required when accessing absolute URL resources where the protocol is not HTTP, and where the encoding is not ISO-8859-1.

Also, when dealing with relative URLs and the HTTP protocol, if the target resource declares a content encoding but proceeds to write a character invalid in that encoding, the treatment of that character is undefined.

---

1. If the responsibility was left to the consumer tag, this could lead to resource leaks (e.g. connection left open, memory space for buffers) until garbage collection is activated. This is because a consumer tag might not close the `Reader`, or because the page author might remove the consumer tag while leaving inadvertantly the <c:import> tag in the page.

2. This restriction could eventually be lifted when the JSP spec supports the notion of page events that actions could register to. On a `pageExit` event, an <c:import> tag would then simply release its resources if it had not already been done, removing the requirement for nested visibility.

Relative and Absolute URLs

The exact semantics of the <c:import> tag depends on what type of URL is being accessed.

Relative URL – same context

This is processed in the exact same way as the include action of the JSP specification (<jsp:include>). The resource belongs to the same web application as the including page and it is specified as a relative URL.

As specified in the JSP specification, a relative URL may either be a context-relative path, or a page-relative path. A *context-relative path* is a path that starts with a "/". It is to be interpreted as relative to the application to which the JSP page belongs. A *page-relative path* is a path that does not start with a "/". It is to be interpreted as relative to the current JSP page, as defined by the rules of inclusion of the <jsp:include> action in the JSP specification.

The semantics of importing a resource specified with a relative URL in the same context are the same as an include performed by a `RequestDispatcher` as defined in the Servlet specification. This means that the whole environment of the importing page is available to the target resource (including request and session attributes, as well as request parameters of the importing page).

Relative URL – foreign context

The resource belongs to a foreign context (web application) hosted under the same container as the importing page. The context name for the resource is specified via attribute `context`.

The relative URL must be context-relative (i.e. must start with a "/") since the including page does not belong to the same context. Similarly, the context name must also start with a "/".

The semantics of importing a resource specified with a relative URL in a foreign context are the same as an include performed by a `RequestDispatcher` on a foreign context as defined in the Servlet specification. This means that only the request environment of the importing page is available to the target resource.

It is important to note that importing resources in foreign contexts may not work in all containers. A security conscious environment may not allow access to foreign contexts. As a workaround, a foreign context resource can also be accessed using an absolute URL. However, it is more efficient to use a relative URL because the resource is then accessed using `RequestDispatcher` defined by the Servlet API.

Relative URL – query parameter aggregation rules

The query parameter aggregation rules work the same way they do with <jsp:include>; the original parameters are augmented with the new parameters, with new values taking precedence over existing values when applicable. The scope of

the new parameters is the import call; the new parameters (and values) will not apply after the import. The behavior is therefore the same as the one defined for the `include()` method of `RequestDispatcher` in the Servlet specification.

Absolute URL

Absolute URLs are retrieved as defined by the `java.net.URL` and `java.net.URLConnection` classes. The <c:import> action therefore supports at a minimum the protocols offered in the J2SE 1.2 platform for absolute URLs. More protocols can be available to a web application, but this will depend on the the class libraries made available to the web application by the platform the container runs on.

When using an absolute URL to import a resource, none of the current execution environment (e.g. request and session attributes) is made available to the target resource, even if that absolute URL resolves to the same host and context path. Therefore, the request parameters of the importing page are not propagated to the target absolute URL.

When importing an external resource using the HTTP protocol, <c:import> behaves according to the semantics of a GET request sent via the `java.net.HttpURLConnection` class, with `setFollowRedirects` set to true.

7.5 \<c:url\>

Builds a URL with the proper rewriting rules applied.

Syntax

Syntax 1: Without body content

```
<c:url value="value" [context="context"]
     [var="varName"] [scope="{page|request|session|application}"]/>
```

Syntax 2: With body content to specify query string parameters

```
<c:url value="value" [context="context"]
       [var="varName"] [scope="{page|request|session|application}"]>
    <c:param> subtags
</c:url>
```

Body Content

JSP. The JSP container processes the body content, then the action trims it and processes it further.

Attributes

Name	Dynamic	Type	Description
value	true	String	URL to be processed.
context	true	String	Name of the context when specifying a relative URL resource that belongs to a foreign context.
var	false	String	Name of the exported scoped variable for the processed url. The type of the scoped variable is String.
scope	false	String	Scope for var.

Description

\<c:url\> processes a URL and rewrites it if necessary. Only relative URLs are rewritten. Absolute URLs are not rewritten to prevent situations where an external URL could be rewritten and expose the session ID. A consequence is that if a page author wants session tracking, only relative URLs must be used with \<c:url\> to link to local resources.

The rewriting must be performed by calling method `encodeURL()` of the Servlet API.

If the URL contains characters that should be encoded (e.g. space), it is the user's responsibility to encode them.

The URL must be either an absolute URL starting with a scheme (e.g. "http://server/context/page.jsp") or a relative URL as defined by JSP 1.2 in JSP.2.2.1 "Relative URL Specification". As a consequence, an implementation must prepend the context path to a URL that starts with a slash (e.g. "/page2.jsp") so that such URLs can be properly interpreted by a client browser.

Specifying a URL in a foreign context is possible through the `context` attribute. The URL specified must must start with a / (since this is a context-relative URL). The context name must also start with a / (since this is a standard way to identify a context).

Because the URL built by this action may include session information as a path parameter, it may fail if used with `RequestDispatcher` of the Servlet API. The consumer of the rewritten URL should therefore remove the session ID information prior to calling `RequestDispatcher`. This situation is properly handled in <c:import>.

By default, the result of the URL processing is written to the current `JspWriter`. It is also possible to export the result as a JSP scoped variable defined via the `var` and `scope` attributes.

<c:param> subtags can also be specified within the body of <c:url> for adding to the URL query string parameters, which will be properly encoded if necessary.

7.6   \<c:redirect\>

Sends an HTTP redirect to the client.

Syntax

Syntax 1: Without body content
```
<c:redirect url="value" [context="context"]/>
```

Syntax 2: With body content to specify query string parameters
```
<c:redirect url="value" [context="context"]/>
    <c:param> subtags
</c:redirect>
```

Body Content

JSP. The JSP container processes the body content, then the action trims it and processes it further.

Attributes

Name	Dyn	Type	Description
url	true	String	The URL of the resource to redirect to.
context	true	String	Name of the context when redirecting to a relative URL resource that belongs to a foreign context.

Description

This action sends an HTTP redirect response to the client and aborts the processing of the page by returning SKIP_PAGE from doEndTag().

The URL must be either an absolute URL starting with a scheme (e.g. "http://server/context/page.jsp") or a relative URL as defined by JSP 1.2 in JSP.2.2.1 "Relative URL Specification". As a consequence, an implementation must prepend the context path to a URL that starts with a slash (e.g. "/page2.jsp") if the behavior is implemented using the HttpServletResponse.sendRedirect() method.

Redirecting to a resource in a foreign context is possible through the context attribute. The URL specified must must start with a "/" (since this is a context-relative URL). The context name must also start with a "/" (since this is a standard way to identify a context).

<c:redirect> follows the same rewriting rules as defined for <c:url>.

7.7 <c:param>

Adds request parameters to a URL. Nested action of <c:import>, <c:url>,
<c:redirect>.

Syntax

Syntax 1: Parameter value specified in attribute "value"
```
<c:param name="name" value="value"/>
```

Syntax 2: Parameter value specified in the body content
```
<c:param name="name">
    parameter value
</c:param>
```

Body Content

JSP. The JSP container processes the body content, then the action trims it and
processes it further.

Attributes

Name	Dynamic	Type	Description
name	true	String	Name of the query string parameter.
value	true	String	Value of the parameter.

Null & Error Handling

- If `name` is null or empty, no action is performed. It is not an error.
- If `value` is null, it is processed as an empty value.

Description

Nested action of <c:import>, <c:url>, <c:redirect> to add request parameters to a
URL. <c:param> also URL encodes both `name` and `value`.

One might argue that this is redundant given that a URL can be constructed to directly specify the query string parameters. For example:

```
<c:import url="/exec/doIt">
    <c:param name="action" value="register"/>
</c:import>
```

is the same as:

```
<c:import url="/exec/doIt?action=register"/>
```

It is indeed redundant, but is consistent with <jsp:include>, which supports nested <jsp:param> sub-elements. Moreover, it has been designed such that the attributes name and value are automatically URL encoded.

Internationalization (i18n) Actions

I18n-capable formatting tag library

With the explosion of application development based on web technologies, and the deployment of such applications on the Internet, applications must be able to adapt to the languages and formatting conventions of their clients. This means that page authors must be able to tailor page content according to the client's language and cultural formatting conventions. For example, the number 345987.246 should be formatted as 345 987,246 for France, 345.987,246 for Germany, and 345,987.246 for the U.S.

The process of designing an application (or page content) so that it can be adapted to various languages and regions without requiring any engineering changes is known as *internationalization*, or i18n for short. Once a web application has been internationalized, it can be adapted for a number of regions or languages by adding locale-specific components and text. This process is known as *localization*.

There are two approaches to internationalizing a web application:

■ Provide a version of the JSP pages in each of the target locales and have a controller servlet dispatch the request to the appropriate page (depending on the requested locale). This approach is useful if large amounts of data on a page or an entire web application need to be internationalized.

■ Isolate any locale-sensitive data on a page (such as error messages, string literals, or button labels) into resource bundles, and access the data via i18n actions, so that the corresponding translated message is fetched automatically and inserted into the page.

The JSTL i18n-capable formatting actions support either approach: They assist page authors with creating internationalized page content that can be localized into any locale available in the JSP container (this addresses the second approach), and allow various data elements such as numbers, currencies, dates and times to be formatted and parsed in a locale-sensitive or customized manner (this may be used in either approach).

JSTL's i18n actions are covered in this chapter. The formatting actions are covered in Chapter 9.

8.1 Overview

There are three key concepts associated with internationalization: locale, resource bundle, and basename.

A *locale* represents a specific geographical, political, or cultural region. A locale is identified by a language code, along with an optional country code[1].

- Language code

  The language code is the lower-case two-letter code as defined by ISO-639 (e.g. "ca" for Catalan, "zh" for Chinese). The full list of these codes can be found at a number of sites, such as:
  http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt

- Country code

  The country code is the upper-case two-letter code as defined by ISO-3166 (e.g. "IT" for Italy, "CR" for Costa Rica). The full list of these codes can be found at a number of sites, such as:
  http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html.

A *resource bundle* contains locale-specific objects. Each message in a resource bundle is associated with a key. Since the set of messages contained in a resource bundle can be localized for many locales, the resource bundles that translate the same set of messages are identified by the same *basename*. A specific resource bundle is therefore uniquely identified by combining its basename with a locale.

For instance, a web application could define the registration resource bundles with basename `Registration` to contain the messages associated with the registration portion of the application. Assuming that French and English are the only languages supported by the application, there will be two resource bundles: `Registration_fr` (French language) and `Registration_en` (English language). Depending on the locale associated with the client request, the key "greeting" could be mapped to the message "Bonjour" (French) or "Hello" (English).

---

1. A variant code may also be specified, although rarely used.

8.1.1   \<fmt:message\>

It is possible to internationalize the JSP pages of a web application simply by using the \<fmt:message\> action as shown below:

```
<fmt:message key="greeting"/>
```

In this case, \<fmt:message\> leverages the default i18n localization context, making it extremely simple for a page author to internationalize JSP pages.

\<fmt:message\> also supports compound messages, i.e. messages that contain one or more variables. Parameter values for these variables may be supplied via one or more \<fmt:param\> subtags (one for each parameter value). This procedure is referred to as *parametric replacement*.

```
<fmt:message key="athletesRegistered">
  <fmt:param>
    <fmt:formatNumber value="${athletesCount}"/>
  </fmt:param>
</fmt:message>
```

Depending on the locale, this example could print the following messages:

```
    french: Il y a 10 582 athletes enregistres.
   english: There are 10,582 athletes registered.
```

8.2   I18n Localization Context

I18n actions use an i18n localization context to localize their data. An i18n localization context contains two pieces of information: a resource bundle and the locale for which the resource bundle was found.

An i18n action determine its i18n localization context in one of several ways, which are described in order of precedence:

- \<fmt:message\> `bundle` attribute

  If attribute `bundle` is specified in \<fmt:message\>, the i18n localization context associated with it is used for localization.

- \<fmt:bundle\> action

If <fmt:message> actions are nested inside a <fmt:bundle> action, the i18n localization context of the enclosing <fmt:bundle> action is used for localization. The <fmt:bundle> action determines the resource bundle of its i18n localization context according to the resource bundle determination algorithm in Section 8.3, using the basename attribute as the resource bundle basename.

■ I18n default localization context

The i18n localization context whose resource bundle is to be used for localization is specified via the `javax.servlet.jsp.jstl.fmt.localizationContext` configuration setting (see Section 8.11.3). If the configuration setting is of type `LocalizationContext`, its resource bundle component is used for localization. Otherwise, the configuration setting is of type `String`, and the action establishes its own i18n localization context whose resource bundle component is determined according to the resource bundle determination algorithm in Section 8.3, using the configuration setting as the resource bundle basename.

The example below shows how the various localization contexts can be established to define the resource bundle used for localization.

```
<%-- Use configuration setting --%>
<fmt:message key="Welcome" />

<fmt:setBundle basename="Errors" var="errorBundle" />
<fmt:bundle basename="Greetings">
  <%-- Localization context established by
       parent <fmt:bundle> tag --%>
  <fmt:message key="Welcome" />
  <%-- Localization context established by attribute bundle --%>
  <fmt:message key="WrongPassword" bundle="${errorBundle}" />
</fmt:bundle>
```

8.2.1   Preferred Locales

If the resource bundle of an i18n localization context needs to be determined, it is retrieved from the web application's resources according to the algorithm described in section Section 8.3. This algorithm requires two pieces of information: the basename of the resource bundle (as described in the previous section) and the preferred locales.

The method for setting the preferred locales is characterized as either application-based or browser-based.

Application-based locale setting has priority over browser-based locale setting. In this mode, the locale is set via the `javax.servlet.jsp.jstl.fmt.locale` configuration setting (see Section 8.11.1). Setting the locale this way is useful in

situations where an application lets its users pick their preferred locale and then sets the scoped variable accordingly. This may also be useful in the case where a client's preferred locale is retrieved from a database and installed for the page using the <fmt:setLocale> action.

The <fmt:setLocale> action may be used to set the `javax.servlet.jsp.jstl.fmt.locale` configuration variable as follows:

```
<fmt:setLocale value="en_US" />
```

In the *browser-based* locale setting, the client determines via its browser settings which locale(s) should be used by the web application. The action retrieves the client's locale preferences by calling `ServletRequest.getLocales()` on the incoming request. This returns a list of the locales (in order of preference) that the client wants to use.

Whether application- or browser-based locale setting is used, an ordered list of preferred locales is fed into the algorithm described in section Section 8.3 to determine the resource bundle for an i18n localization context.

8.3 Determinining the Resource Bundle for an i18n Localization Context

Given a basename and an ordered set of preferred locales, the resource bundle for an i18n localization context is determined according to the algorithm described in this section.

8.3.1 Resource Bundle Lookup

Localization in JSTL is based on the same mechanisms offered in the J2SE platform. Resource bundles contain locale-specific objects, and when an i18n action requires a locale-specific resource, it simply loads it from the appropriate resource bundle.

The algorithm of Section 8.3.2 describes how the proper resource bundle is determined. This algorithm calls for a resource bundle lookup, where an attempt is made at fetching a resource bundle associated with a specific basename and locale.

JSTL leverages the semantics of the `java.util.ResourceBundle` method

```
getBundle(String basename, java.util.Locale locale)
```

for resource bundle lookup, with one important modification.

As stated in the documentation for `ResourceBundle`, a resource bundle lookup searches for classes and properties files with various suffixes on the basis of:

1. The specified locale
2. The current default locale as returned by `Locale.getDefault()`
3. The root resource bundle (basename)

In JSTL, the search is limited to the first level; i.e. the specified locale. Steps 2 and 3 are removed so that other locales may be considered before applying the JSTL fallback mechanism described in Section 8.3.2. Only if no fallback mechanism exists, or the fallback mechanism fails to determine a resource bundle, is the root resource bundle considered.

Resource bundles are therefore searched in the following order:

basename + "_" + language + "_" + country + "_" + variant
basename + "_" + language + "_" + country
basename + "_" + language

8.3.2   Resource Bundle Determination Algorithm

Notes:

- When there are multiple preferred locales, they are processed in the order they were returned by `ServletRequest.getLocales()`.
- The algorithm stops as soon as a resource bundle has been selected for the localization context.

Step 1: Find a match within the ordered set of preferred locales

A resource bundle lookup (see Section 8.3.1) is performed for each one of the preferred locales until a match is found. If a match is found, the locale that led to the match and the matched resource bundle are stored in the i18n localization context.

Step 2: Find a match with the fallback locale

A resource bundle lookup (see Section 8.3.1) is performed for the fallback locale specified in the `javax.servlet.jsp.jstl.fmt.fallbackLocale` configuration setting. If a match is found, the fallback locale and the matched resource bundle are stored in the i18n localization context.

If no match is found following the above two steps, an attempt is made to load the root resource bundle with the given basename. If such a resource bundle exists, it is used as the resource bundle of an i18n localization context that does not have any locale. Otherwise, the established i18n localization context contains neither a resource bundle nor a locale. It is then up to the i18n action relying on this i18n localization context for the localization of its data to take a proper corrective action.

It is important to note that this algorithm gives higher priority to a *language* match over an *exact* match that would have occurred further down the list of preferred locales. For example, if the browser-based locale settings are "en" and "fr_CA", with resource bundles "Messages_en" and "Messages_fr_CA", the Messages_en bundle will be selected as the resource bundle for the localization context.

The definition of a fallback locale along with its associated resource bundles is the only portable way a web application can ensure the proper localization of all its internationalized pages. The algorithm of this section never considers the default locale associated with the Java runtime of the container because this would result in a non-portable behavior.

The behavior is implementation-specific if the set of available resource bundles changes during execution of the page. Implementations may thus cache whatever information they deem necessary to improve the performance of the algorithm presented in this section.

8.3.3   Examples

The following examples demonstrate how the resource bundle is determined for an i18n localization context.

Example 1

Settings
Basename: `Resources`
Ordered preferred locales: `en_GB, fr_CA`
Fallback locale: fr_CA
Resource bundles: `Resources_en, Resources_fr_CA`

Algorithm Trace
Step 1: Find a match within the ordered set of preferred locales
  `en_GB`   match with `Resources_en`

Result
Resource bundle selected: `Resources_en`
Locale: `en`

Example 2

Settings
Basename: `Resources`
Ordered preferred locales: `de, fr`
Fallback locale: `en`
Resource bundles: `Resources_en`

Algorithm Trace
Step 1: Find a match within the ordered set of preferred locales
  `de`   no match
  `fr`   no match
Step 2: Find a match with the fallback locale
  `en`   exact match with `Resources_en`

Result
Resource bundle selected: `Resources_en`
Locale: `en`

Example 3

Settings
Basename: `Resources`
Ordered preferred locales: `ja, en_GB, en_US, en_CA, fr`
Fallback locale: `en`
Resource bundles: `Resources_en, Resources_fr, Resources_en_US`

Algorithm Trace
Step 1: Find a match within the ordered set of preferred locales

```
    ja         no match
    en_GB       match with Resources_en
```

<u>Result</u>
Resource bundle selected: `Resources_en`
Locale: en

Example 4

<u>Settings</u>
Basename: `Resources`
Ordered preferred locales: `fr, sv`
Fallback locale: `en`
Resource bundles: `Resources_fr_CA, Resources_sv, Resources_en`

<u>Algorithm Trace</u>
Step 1: Find a match within the ordered set of preferred locales
```
    fr         no match
    sv          match with Resources_sv
```

<u>Result</u>
Resource bundle selected: `Resources_sv`
Locale: sv

This example shows that whenever possible, a resource bundle for a specific language and country (`Resources_fr_CA`) should be backed by a resource bundle covering just the language (`Resources_fr`). If the country-specific differences of a language are too significant for there to be a language-only resource bundle, it is expected that clients will specify both a language and a country as their preferred language, in which case an exact resource bundle match will be found.

8.4   Response Encoding

Any i18n action that establishes a localization context is responsible for setting the response's locale of its page, unless the localization context that was established does not have any locale. This is done by calling method `ServletResponse.setLocale()` with the locale of the localization context.

This assumes that the response is buffered with a big enough buffer size, since `ServletResponse.setLocale()` and `ServletResponse.setContentType()` must be called before `ServletResponse.getWriter()` in order for the specified locale or charset to affect the construction of the writer.[1]

---

1. It also assumes that the JSP container doesn't call `getWriter()` until it's about to flush the buffer. An errata has been published for the JSP 1.2 specification to clarify this.

More specifically, the response's `setLocale()` method is always called by the <fmt:setLocale> action (see Section 8.5). In addition, it is called by the following actions:

- Any <fmt:bundle> (see Section 8.6) and <fmt:setBundle> (see Section 8.7) action.
- Any <fmt:message> action that establishes an i18n localization context
- Any formatting action that establishes a formatting locale on its own (see Section 9.3).

After an action has called `ServletResponse.setLocale()`, and sessions are enabled, it must determine the character encoding associated with the response locale (by calling `ServletResponse.getCharacterEncoding()`) and store it in the scoped variable `javax.servlet.jsp.jstl.fmt.request.charset` in session scope. This attribute may be used by the <fmt:requestEncoding> action (see Section 8.10) in a page invoked by a form included in the response to set the request charset to the same as the response charset. This makes it possible for the container to decode the form parameter values properly, since browsers typically encode form field values using the response's charset.

8.5   \<fmt:setLocale>

Stores the specified locale in the `javax.servlet.jsp.jstl.fmt.locale` configuration variable.

Syntax

```
<fmt:setLocale value="locale"
          [variant="variant"]
          [scope="{page|request|session|application}"]/>
```

Body Content

Empty.

Attributes

Name	Dynamic	Type	Description
value	true	String or java.util.Locale	A `String` value is interpreted as the printable representation of a locale, which must contain a two-letter (lower-case) language code (as defined by ISO-639), and may contain a two-letter (upper-case) country code (as defined by ISO-3166). Language and country codes must be separated by hyphen ('-') or underscore ('_').
variant	true	String	Vendor- or browser-specific variant. See the `java.util.Locale` javadocs for more information on variants.
scope	false	String	Scope of the *locale* configuration variable.

Null & Error Handling

■ If `value` is null or empty, use the runtime default locale.

Description

The <fmt:setLocale> action stores the locale specified by the `value` attribute in the `javax.servlet.jsp.jstl.fmt.locale` configuration variable in the scope given by the `scope` attribute. If `value` is of type `java.util.Locale`, `variant` is ignored.

As a result of using this action, browser-based locale setting capabilities are disabled. This means that if this action is used, it should be declared at the beginning of a page, before any other i18n-capable formatting actions.

8.6   <fmt:bundle>

Creates an i18n localization context to be used by its body content.

Syntax

```
<fmt:bundle basename="basename"
            [prefix="prefix"]>
   body content
</fmt:bundle>
```

Body Content

JSP. The JSP container processes the body content and then writes it to the current `JspWriter.` The action ignores the body content.

Attributes

Name	Dynamic	Type	Description
basename	true	String	Resource bundle base name. This is the bundle's fully-qualified resource name, which has the same form as a fully-qualified class name, that is, it uses "." as the package component separator and does not have any file type (such as ".class" or ".properties") suffix.
prefix	true	String	Prefix to be prepended to the value of the message key of any nested <fmt:message> action.

Null & Error Handling

- If `basename` is null or empty, or a resource bundle cannot be found, the `null` resource bundle is stored in the i18n localization context.

Description

The <fmt:bundle> action creates an i18n localization context and loads its resource bundle into that context. The name of the resource bundle is specified with the `basename` attribute.

The specific resource bundle that is loaded is determined according to the algorithm presented in Section 8.3.2.

The scope of the i18n localization context is limited to the action's body content.

The `prefix` attribute is provided as a convenience for very long message key names. Its value is prepended to the value of the message `key` of any nested <fmt:message> actions.

For example, using the `prefix` attribute, the key names in:

```
<fmt:bundle basename="Labels">
  <fmt:message key="com.acme.labels.firstName"/>
  <fmt:message key="com.acme.labels.lastName"/>
</fmt:bundle>
```

may be abbreviated to:

```
<fmt:bundle basename="Labels" prefix="com.acme.labels.">
  <fmt:message key="firstName"/>
  <fmt:message key="lastName"/>
</fmt:bundle>
```

8.7   <fmt:setBundle>

Creates an i18n localization context and stores it in the scoped variable or the `javax.servlet.jsp.jstl.fmt.localizationContext` configuration variable.

Syntax

```
<fmt:setBundle basename="basename"
               [var="varName"]
               [scope="{page|request|session|application}"]/>
```

Body Content

Empty.

Attributes

Name	Dynamic	Type	Description
basename	true	String	Resource bundle base name. This is the bundle's fully-qualified resource name, which has the same form as a fully-qualified class name, that is, it uses "." as the package component separator and does not have any file type (such as ".class" or ".properties") suffix.
var	false	String	Name of the exported scoped variable which stores the i18n localization context of type `javax.servlet.jsp.jstl.fmt.LocalizationContext`.
scope	false	String	Scope of `var` or the localization context configuration variable.

Null & Error Handling

■ If `basename` is null or empty, or a resource bundle cannot be found, the `null` resource bundle is stored in the i18n localization context.

Description

The <fmt:setBundle> action creates an i18n localization context and loads its resource bundle into that context. The name of the resource bundle is specified with the `basename` attribute.

The specific resource bundle that is loaded is determined according to the algorithm presented in Section 8.3.2.

The i18n localization context is stored in the scoped variable whose name is given by `var`. If `var` is not specified, it is stored in the `javax.servlet.jsp.jstl.fmt.localizationContext` configuration variable, thereby making it the new default i18n localization context in the given scope.

8.8   `<fmt:message>`

Looks up a localized message in a resource bundle.

Syntax

Syntax 1: without body content

```
<fmt:message   key="messageKey"
               [bundle="resourceBundle"]
               [var="varName"]
               [scope="{page|request|session|application}"]/>
```

Syntax 2: with a body to specify message parameters

```
<fmt:message key="messageKey"
                [bundle="resourceBundle"]
               [var="varName"]
               [scope="{page|request|session|application}"]>

   <fmt:param> subtags

</fmt:message>
```

Syntax 3: with a body to specify key and optional message parameters

```
<fmt:message [bundle="resourceBundle"]
               [var="varName"]
               [scope="{page|request|session|application}"]>

    key
   optional <fmt:param> subtags
</fmt:message>
```

Body Content

JSP. The JSP container processes the body content, then the action trims it and
processes it further.

Attributes

Name	Dyn	Type	Description
key	true	String	Message key to be looked up.
bundle	true	LocalizationContext	Localization context in whose resource bundle the message key is looked up.
var	false	String	Name of the exported scoped variable which stores the localized message.
scope	false	String	Scope of var.

Constraints

- If scope is specified, var must also be specified.

Null & Error Handling

- If key is null or empty, the message is processed as if undefined; that is, an error message of the form "??????" is produced.
- If the i18n localization context that this action determines does not have any resource bundle, an error message of the form "???<key>???" is produced

Description

The <fmt:message> action looks up the localized message corresponding to the given message key.

The message key may be specified via the key attribute or from the tag's body content. If this action is nested inside a <fmt:bundle> action, and the parent <fmt:bundle> action contains a prefix attribute, the specified prefix is prepended to the message key.

<fmt:message> uses the resource bundle of the i18n localization context determined according to Section 8.2.

If the given key is not found in the resource bundle, or the i18n localization context does not contain any resource bundle, the result of the lookup is an error message of the form "???<key>???" (where <key> is the name of the undefined message key).

If the message corresponding to the given key is *compound*, that is, contains one or more variables, it may be supplied with parameter values for these variables via one or more <fmt:param> subtags (one for each parameter value). This procedure is referred to as *parametric replacement*. Parametric replacement takes place in the order of the <fmt:param> subtags.

In the presence of one or more <fmt:param> subtags, the message is supplied to the java.text.MessageFormat method applyPattern(), and the values of the <fmt:param> tags are collected in an Object[] and supplied to the java.text.MessageFormat method format(). The locale of the java.text.MessageFormat is set to the appropriate localization context locale before applyPattern() is called.

If the message is compound and no <fmt:param> subtags are specified, it is left unmodified (that is, java.text.MessageFormat is not used).

The <fmt:message> action outputs its result to the current JspWriter object, unless the var attribute is specified, in which case the result is stored in the named JSP attribute.

8.9   \<fmt:param\>

Supplies a single parameter for parametric replacement to a containing
\<fmt:message\> (see Section 8.8) action.

Syntax

Syntax 1: value specified via attribute "value"
```
<fmt:param value="messageParameter"/>
```

Syntax 2: value specified via body content
```
<fmt:param>
   body content
</fmt:param>
```

Body Content

JSP. The JSP container processes the body content, then the action trims it and
processes it further.

Attributes

Name	Dynamic	Type	Description
value	true	Object	Argument used for parametric replacement.

Constraints

■ Must be nested inside a \<fmt:message\> action.

Description

The \<fmt:param\> action supplies a single parameter for parametric replacement to
the compound message given by its parent \<fmt:message\> action.

Parametric replacement takes place in the order of the <fmt:param> tags. The semantics of the replacement are defined as in the class `java.text.MessageFormat`:

the compound message given by the parent <fmt:message> action is used as the argument to the `applyPattern()` method of a `java.text.MessageFormat` instance, and the values of the <fmt:param> tags are collected in an `Object[]` and supplied to that instance's `format()` method.

The argument value may be specified via the `value` attribute or inline via the tag's body content.

8.10   <fmt:requestEncoding>

Sets the request's character encoding.

Syntax

```
<fmt:requestEncoding [value="charsetName"]/>
```

Body Content

Empty.

Attributes

Name	Dynamic	Type	Description
value	true	String	Name of character encoding to be applied when decoding request parameters.

Description

The <fmt:requestEncoding> action may be used to set the request's character encoding, in order to be able to correctly decode request parameter values whose encoding is different from ISO-8859-1.

This action is needed because most browsers do not follow the HTTP specification and fail to include a `Content-Type` header in their requests.

More specifically, the purpose of the <fmt:requestEncoding> action is to set the request encoding to be the same as the encoding used for the response containing the form that invokes this page. This is needed because even if the encoding of the page that is generating the response containing the form is specified via the `contentType` attribute of a page directive, the response's actual locale (and thus character encoding) may differ from the value specified in the page directive if the page contains an i18n-capable formatting action which sets the response's locale (and thus character encoding) by calling `ServletResponse.setLocale()`, thereby overriding any encoding specified in the page directive (see Section 8.4). This assumes that the JSP container calls `ServletResponse.setContentType()` (with the value of the page directive's `contentType` attribute, if present) at the beginning of the page, and that any subsequent call to `ServletResponse.setLocale()` (by any i18n-capable formatting action in the

page) overrides the page's existing encoding, so that the locale (and encoding) set by the last i18n-capable formatting action in the page wins. (This assumes that the response is buffered with a big enough buffer size, since `ServletResponse.setLocale()` and `ServletResponse.setContentType()` must be called before `ServletResponse.getWriter()` in order for the specified locale or charset to affect the construction of the writer.)

This action calls the `setCharacterEncoding()` method on the servlet request with the character encoding name specified in the `value` attribute. It must be used before any parameters are retrieved, either explicitly or through the use of an EL expression.

If the character encoding of the request parameters is not known in advance (since the locale and thus character encoding of the page that generated the form collecting the parameter values was determined dynamically), the `value` attribute must not be specified. In this case, the <fmt:requestEncoding> action first checks if there is a charset defined in the request `Content-Type` header. If not, it uses the character encoding from the `javax.servlet.jsp.jstl.fmt.request.charset` scoped variable which is searched in session scope. If this scoped variable is not found, the default character encoding (ISO-8859-1) is used.

8.11 Configuration Settings

This section describes the i18n-related configuration settings. Refer to Section 2.8 for more information on how JSTL processes configuration data.

8.11.1 Locale

Variable name	javax.servlet.jsp.jstl.fmt.locale
Java Constant	Config.FMT_LOCALE
Type	String or java.util.Locale
Set by	<fmt:setLocale>
Used by	<fmt:bundle>, <fmt:setBundle>, <fmt:message>, <fmt:formatNumber>, <fmt:parseNumber>, <fmt:formatDate>, <fmt:parseDate>

Specifies the locale to be used by the i18n-capable formatting actions, thereby disabling browser-based locales. A String value is interpreted as defined in action <fmt:setLocale> (see Section 8.5).

8.11.2 Fallback Locale

Variable name	javax.servlet.jsp.jstl.fmt.fallbackLocale
Java Constant	Config.FMT_FALLBACK_LOCALE
Type	String or java.util.Locale
Set by	
Used by	<fmt:bundle>, <fmt:setBundle>, <fmt:message>, <fmt:formatNumber>, <fmt:parseNumber>, <fmt:formatDate>, <fmt:parseDate>

Specifies the fallback locale to be used by the i18n-capable formatting actions if none of the preferred match any of the available locales. A String value is interpreted as defined in action <fmt:setLocale> (see Section 8.5).

8.11.3   I18n Localization Context

Variable name	`javax.servlet.jsp.jstl.fmt.localizationContext`
Java Constant	`Config.FMT_LOCALIZATION_CONTEXT`
Type	`String` or `javax.servlet.jsp.jstl.fmt.LocalizationContext`
Set by	`<fmt:setBundle>`
Used by	`<fmt:message>`, `<fmt:formatNumber>`, `<fmt:parseNumber>`, `<fmt:formatDate>`, `<fmt:parseDate>`

Specifies the default i18n localization context to be used by the i18n-capable formatting actions. A `String` value is interpreted as a resource bundle basename.

8.12 Class LocaleSupport

The locale-determination logic for resource bundles, described in Section 8.3, is exposed as a general convenience method in the `LocaleSupport` class so that it may be used by any tag handler implementation that needs to produce localized messages. For example, this is useful for exception messages that are intended directly for user consumption on an error page.

Method `getLocalizedMessage()` looks up a given message key in the resource bundle with a given base name (or the resource bundle of the default i18n localization context) and optionally performs parametric replacement on the result of the lookup before returning it.

Syntax

```
javax.servlet.jsp.jstl.fmt
public class LocaleSupport
```

Methods

```
public static String getLocalizedMessage(PageContext pc, String key)
```

Retrieves the localized message corresponding to the given key. The given key is looked up in the resource bundle of the default i18n localization context, which is retrieved from the `javax.servlet.jsp.jstl.fmt.localizationContext` configuration setting. If the configuration setting is empty, or the default i18n localization context does not contain any resource bundle, or the given key is undefined in its resource bundle, the string "???<key>???" is returned, where "<key>" is replaced with the given key.

Returns
  The localized message corresponding to the given key.

```
public static String getLocalizedMessage(PageContext pc, String key,
String basename)
```

Retrieves the localized message corresponding to the given key. The given key is looked up in the resource bundle with the given base name. If no resource bundle with the given base name exists, or the given key is undefined in the resource bundle, the string "???<key>???" is returned, where "<key>" is replaced with the given key.

Returns
  The localized message corresponding to the given key.

```
public static String getLocalizedMessage(PageContext pc, String key,
                                          Object[] args)
```

Retrieves the localized message corresponding to the given key and performs parametric replacement using the arguments specified via `args`. See the description of the <fmt:message> action in Section 8.8 for a description of how parametric replacement is implemented. The localized message is retrieved as in method getLocalizedMessage(PageContext pc, String key).

Returns

  The localized message corresponding to the given key.

```
public static String getLocalizedMessage(PageContext pc, String key,
                                          Object[] args, String basename)
```

Retrieves the localized message corresponding to the given key and performs parametric replacement using the arguments specified via `args`. See the description of the <fmt:message> action in Section 8.8 for a description of how parametric replacement is implemented. The localized message is retrieved as in method `getLocalizedMessage(PageContext pc, String key, String basename)`.

Returns

  The localized message corresponding to the given key.

8.13 Class LocalizationContext

This class represents an i18n localization context, which has two components: a resource bundle and a locale.

The resource bundle component is used by <fmt:message> for mapping message keys to localized messages, and the locale component is used by the <fmt:formatNumber>, <fmt:parseNumber>, <fmt:formatDate>, and <fmt:parseDate> actions as their formatting or parsing locale, respectively.

Syntax

```
javax.servlet.jsp.jstl.fmt
public class LocalizationContext
```

Constructors

public **LocalizationContext**()

Constructs an empty i18n localization contex.

public **LocalizationContext**(ResourceBundle bundle)

Constructs an I18N localization context from the given resource bundle.

public **LocalizationContext**(ResourceBundle bundle, Locale locale)

Constructs an I18N localization context from the given resource bundle and locale.

Methods

public java.util.ResourceBundle **getResourceBundle**()

Gets the resource bundle of this i18n localization context.

Returns

The resource bundle of this i18n localization context, or null if this i18n localization context is empty (i.e. instantiated with the empty constructor).

public java.util.Locale **getLocale**()

Gets the locale of this i18n localization context.

Returns

The locale of this i18n localization context, or null if this i18n localization context is empty (i.e. instantiated with the empty constructor), or its resource bundle is a (locale-less) root resource bundle.

Formatting Actions
I18n-capable formatting tag library

The JSTL formatting actions allow various data elements in a JSP page, such as numbers, dates and times, to be formatted and parsed in a locale-sensitive or customized manner.

9.1 Overview

9.1.1 Formatting Numbers, Currencies, and Percentages

The <fmt:formatNumber> action allows page authors to format numbers, currencies, and percentages according to the client's cultural formatting conventions.

For example, the output of:

```
<fmt:formatNumber value="9876543.21" type="currency"/>
```

varies with the page's locale (given in parentheses), as follows:

```
SFr. 9'876'543.21 (fr_CH)
$9,876,543.21 (en_US)
```

While the previous example uses the default formatting pattern (for currencies) of the page's locale, it is also possible to specify a customized formatting pattern. For example, a pattern of ".000" will cause any numeric value formatted with it to be represented with 3 fraction digits, adding trailing zeros if necessary, so that:

```
<fmt:formatNumber value="12.3" pattern=".000"/>
```

will output "12.300".

Likewise, a pattern of "#,#00.0#" specifies that any numeric value formatted with it will be represented with a minimum of 2 integer digits, 1 fraction digit, and a maximum of 2 fraction digits, with every 3 integer digits grouped. Applied to "123456.7891", as in:

```
<fmt:formatNumber value="123456.7891" pattern="#,#00.0#"/>
```

the formatted output will be "123,456.79" (note that rounding is handled automatically).

The following example formats a numeric value as a currency, stores it in a scoped variable, parses it back in, and outputs the parsed result (which is the same as the original numeric value):

```
<fmt:formatNumber value="123456789" type="currency" var="cur"/>
<fmt:parseNumber value="${cur}" type="currency"/>
```

A similar sequence of actions could have been used to retrieve a currency-formatted value from a database, parse its numeric value, perform an arithmetic operation on it, reformat it as a currency, and store it back to the database.

9.1.2   Formatting Dates and Times

The <fmt:formatDate> action allows page authors to format dates and times according to the client's cultural formatting conventions.

For example, assuming a current date of Oct 22, 2001 and a current time of 4:05:53PM, the following action:

```
<jsp:useBean id="now" class="java.util.Date" />
<fmt:formatDate value="${now}" timeStyle="long"
dateStyle="long"/>
```

will output

```
October 22, 2001 4:05:53 PM PDT
```

for the U.S. and

```
22 octobre 2001 16:05:53 GMT-07:0
```

for the French locale.

Page authors may also specify a customized formatting style for their dates and times. Assuming the same current date and time as in the above example, this action:

```
<fmt:formatDate value="${now}" pattern="dd.MM.yy"/>
```

will output

```
22.10.01
```

for the U.S. locale.

Time information on a page may be tailored to the preferred time zone of a client. This is useful if the server hosting the page and its clients reside in different time zones. If time information is to be formatted or parsed in a time zone different from that of the JSP container, the <fmt:formatDate> and <fmt:parseDate> action may be nested inside a <fmt:timeZone> action or supplied with a timeZone attribute.

In the following example, the current date and time are formatted in the "GMT+1:00" time zone:

```
<fmt:timeZone value="GMT+1:00">
  <fmt:formatDate value="${now}" type="both" dateStyle="full"
timeStyle="full"/>
</fmt:timeZone>
```

9.2 Formatting Locale

A formatting action[1] may leverage an i18n localization context to determine its formatting locale or establish a formatting locale on its own, by following these steps:

1. Four formatting actions localize their data: <fmt:formatNumber>, <fmt:parseNumber>, <fmt:formatDate>, <fmt:parseDate>.

- <fmt:bundle> action

  If a formatting action is nested inside a <fmt:bundle> action (see Section 8.6), the locale of the i18n localization context of the enclosing <fmt:bundle> action is used as the formatting locale. The <fmt:bundle> action determines the resource bundle of its i18n localization context according to the resource bundle determination algorithm in Section 8.3, using the basename attribute as the resource bundle basename. If the i18n localization context of the enclosing <fmt:bundle> action does not contain any locale, go to the next step.

- I18n default localization context

  The default i18n localization context may be specified via the `javax.servlet.jsp.jstl.fmt.localizationContext` configuration setting. If such a configuration setting exists, and its value is of type `LocalizationContext`, its locale is used as the formatting locale. Otherwise, if the configuration setting is of type `String`, the formatting action establishes its own i18n localization context and uses its locale as the formatting locale (in this case, the resource bundle component of the i18n localization context is determined according to the resource bundle determination algorithm in Section 8.3, using the configuration setting as the resource bundle basename). If the i18n localization context determined in this step does not contain any locale, go to the next step.

- Formatting locale lookup

  The formatting action establishes a locale according to the algorithm described in Section 9.3. This algorithm requires the preferred locales. The way the preferred locales are set is exactly the same as with i18n actions and is described in Section 8.2.1.

The following example shows how the various localization contexts can be established to define the formatting locale.

```
<jsp:useBean id="now" class="java.util.Date" />

<%-- Formatting locale lookup --%>
<fmt:formatDate value="${now}" />

<fmt:bundle basename="Greetings">
  <%-- I18n localization context from parent <fmt:bundle> tag --%>
  <fmt:message key="Welcome" />
  <fmt:formatDate value="${now}" />
</fmt:bundle>
```

9.3 Establishing a Formatting Locale

If a formatting action fails to leverage an i18n localization context for its formatting locale – either because the formatting action has no way of referring to an i18n localization context, or the i18n localization context does not have any locale – it must establish the formatting locale on its own, given an ordered set of preferred locales, according to the formatting locale lookup algorithm described in this section.

9.3.1 Locales Available for Formatting Actions

The algorithm described in Section 9.3.3 compares preferred locales against the set of locales that are available for a specific formatting action.

The locales available for actions <fmt:formatNumber> and <fmt:parseNumber> are determined by a call to `java.text.NumberFormat.getAvailableLocales()`.

The locales available for <fmt:formatDate> and <fmt:parseDate> are determined by a call to `java.text.DateFormat.getAvailableLocales()`.

9.3.2 Locale Lookup

The algorithm of Section 9.3.3 describes how the proper locale is determined. This algorithm calls for a locale lookup: it attempts to find among the available locales, a locale that matches the specified one.

The locale lookup is similar to the resource bundle lookup described in Section 8.3.1, except that instead of trying to match a resource bundle, the locale lookup tries to find a match in a list of available locales. A match of the specified locale against an available locale is therefore attempted in the following order:

- Language, country, and variant are the same
- Language and country are the same
- Language is the same and the available locale does not have a country

9.3.3 Formatting Locale Lookup Algorithm

Notes:

- When there are multiple preferred locales, they are processed in the order they were returned by a call to `ServletRequest.getLocales()`.

- The algorithm stops as soon as a locale has been selected for the localization context.

Step 1: Find a match within the ordered set of preferred locales

A locale lookup (see Section 9.3.2) is performed for each one of the preferred locales until a match is found. The first match is used as the formatting locale.

Step 2: Find a match with the fallback locale

A locale lookup (see Section 9.3.2) is performed for the fallback locale specified in the `javax.servlet.jsp.jstl.fmt.fallbackLocale` configuration setting. If a match exists, it is used as the formatting locale.

If no match is found after the above two steps, it is up to the formatting action to take a corrective action.

The result of the formatting locale lookup algorithm may be cached, so that subsequent formatting actions that need to establish the formatting locale on their own may leverage it.

9.4   Time Zone

Time information on a page may be tailored to the preferred time zone of a client. This is useful if the server hosting the page and its clients reside in different time zones (page authors could be advised to always use the "long" time format which includes the time zone, but that would still require clients to convert the formatted time into their own time zone).

When formatting time information using the <fmt:formatDate> action (see Section 9.8), or parsing time information that does not specify a time zone using the <fmt:parseDate> action (see Section 9.9), the time zone to use is determined as follows and in this order:

- Use the time zone from the action's `timeZone` attribute.
- If attribute `timeZone` is not specified and the action is nested inside an <fmt:timeZone> action, use the time zone from the enclosing <fmt:timeZone> action.
- Use the time zone given by the `javax.servlet.jsp.jstl.fmt.timeZone` configuration setting.
- Use the JSP container's time zone.

9.5 <fmt:timeZone>

Specifies the time zone in which time information is to be formatted or parsed in its
body content.

Syntax

```
<fmt:timeZone value="timeZone">
   body content
</fmt:timeZone>
```

Body Content

JSP. The JSP container processes the body content and then writes it to the current
`JspWriter`. The action ignores the body content.

Attributes

Name	Dyn	Type	Description
value	true	String or java.util.TimeZone	The time zone. A `String` value is interpreted as a time zone ID. This may be one of the time zone IDs supported by the Java platform (such as "America/Los_Angeles") or a custom time zone ID (such as "GMT-8"). See `java.util.TimeZone` for more information on supported time zone formats.

Null & Error Handling

- If `value` is null or empty, the GMT timezone is used.

Description

The <fmt:timeZone> action specifies the time zone in which to format or parse the
time information of any nested time formatting (see Section 9.8) or parsing (see
Section 9.9) actions.

If the time zone is given as a string, it is parsed using
`java.util.TimeZone.getTimeZone()`.

9.6   `<fmt:setTimeZone>`

Stores the specified time zone in a scoped variable or the time zone configuration
variable.

Syntax

```
<fmt:setTimeZone value="timeZone"
                 [var="varName"]
                 [scope="{page|request|session|application}"]/>
```

Body Content

Empty.

Attributes

Name	Dyn	Type	Description
value	true	`String` or `java.util.TimeZone`	The time zone. A `String` value is interpreted as a time zone ID. This may be one of the time zone IDs supported by the Java platform (such as "America/Los_Angeles") or a custom time zone ID (such as "GMT-8"). See java.util.TimeZone for more information on supported time zone formats.
var	false	`String`	Name of the exported scoped variable which stores the time zone of type `java.util.TimeZone`.
scope	false	`String`	Scope of `var` or the time zone configuration variable.

Null & Error Handling

■ If `value` is null or empty, the GMT timezone is used.

Description

The <fmt:setTimeZone> action stores the given time zone in the scoped variable whose name is given by `var`. If `var` is not specified, the time zone is stored in the `javax.servlet.jsp.jstl.fmt.timeZone` configuration variable, thereby making it the new default time zone in the given scope.
If the time zone is given as a string, it is parsed using `java.util.TimeZone.getTimeZone()`.

9.7   <fmt:formatNumber>

Formats a numeric value in a locale-sensitive or customized manner as a number, currency, or percentage.

Syntax

Syntax 1: without a body

```
<fmt:formatNumber value="numericValue"
                [type="{number|currency|percent}"]
                [pattern="customPattern"]
                [currencyCode="currencyCode"]
                [currencySymbol="currencySymbol"]
                [groupingUsed="{true|false}"]
                [maxIntegerDigits="maxIntegerDigits"]
                [minIntegerDigits="minIntegerDigits"]
                [maxFractionDigits="maxFractionDigits"]
                [minFractionDigits="minFractionDigits"]
                [var="varName"]
                [scope="{page|request|session|application}"]/>
```

Syntax 2: with a body to specify the numeric value to be formatted

```
<fmt:formatNumber [type="{number|currency|percent}"]
                [pattern="customPattern"]
                [currencyCode="currencyCode"]
                [currencySymbol="currencySymbol"]
                [groupingUsed="{true|false}"]
                [maxIntegerDigits="maxIntegerDigits"]
                [minIntegerDigits="minIntegerDigits"]
                [maxFractionDigits="maxFractionDigits"]
                [minFractionDigits="minFractionDigits"]
                [var="varName"]
                [scope="{page|request|session|application}"]>
    numeric value to be formatted
</fmt:formatNumber>
```

Body Content

JSP. The JSP container processes the body content, then the action trims it and processes it further.

Attributes

Name	Dyn	Type	Description
value	true	String or Number	Numeric value to be formatted.
type	true	String	Specifies whether the value is to be formatted as number, currency, or percentage.
pattern	true	String	Custom formatting pattern.
currencyCode	true	String	ISO 4217 currency code. Applied only when formatting currencies (i.e. if type is equal to "currency"); ignored otherwise.
currencySymbol	true	String	Currency symbol. Applied only when formatting currencies (i.e. if type is equal to "currency"); ignored otherwise.
groupingUsed	true	boolean	Specifies whether the formatted output will contain any grouping separators.
maxIntegerDigits	true	int	Maximum number of digits in the integer portion of the formatted output.
minIntegerDigits	true	int	Minimum number of digits in the integer portion of the formatted output.
maxFractionDigits	true	int	Maximum number of digits in the fractional portion of the formatted output.
minFractionDigits	true	int	Minimum number of digits in the fractional portion of the formatted output.
var	false	String	Name of the exported scoped variable which stores the formatted result as a String.
scope	false	String	Scope of var.

Constraints

- If scope is specified, var must also be specified.
- The value of the currencyCode attribute must be a valid ISO 4217 currency code.

Null & Error Handling

- If value is null or empty, remove the scoped variable if it is specified (see attributes var and scope).

- If this action fails to determine a formatting locale, it uses `Number.toString()` as the output format.
- If the attribute `pattern` is null or empty, it is ignored.
- If an exception occurs during the parsing of a string value, it must be caught and rethrown as a `JspException`. The message of the rethrown `JspException` must include the string value, and the caught exception must be provided as the root cause.

Description

The numeric value to be formatted may be specified via the `value` attribute; if missing, it is read from the tag's body content.

The formatting pattern may be specified via the `pattern` attribute, or is looked up in a locale-dependent fashion.

A pattern string specified via the `pattern` attribute must follow the pattern syntax specified by the class `java.text.DecimalFormat`.

If looked up in a locale-dependent fashion, the formatting pattern is determined via a combination of the formatting locale, which is determined according to Section 9.2, and the `type` attribute. Depending on the value of the `type` attribute, the given numeric value is formatted as a number, currency, or percentage. The locale's default formatting pattern for numbers, currencies, or percentages is determined by calling the `java.text.NumberFormat` method `getNumberInstance`, `getCurrencyInstance`, or `getPercentInstance`, respectively, with the formatting locale.

The `pattern` attribute takes precedence over `type`. In either case, the formatting symbols (such as decimal separator and grouping separator) are given by the formatting locale.

The (specified or locale-dependent) formatting pattern may be further fine-tuned using the formatting options described below.

If the numeric value is given as a string literal, it is first parsed into a `java.lang.Number`. If the string does not contain any decimal point, it is parsed using `java.lang.Long.valueOf()`, or `java.lang.Double.valueOf()` otherwise.

The formatted result is output to the current `JspWriter` object, unless the `var` attribute is given, in which case it is stored in the named scoped variable.

Formatting Options

The `groupingUsed` attribute specifies whether the formatted ouput will contain any grouping separators. See the `java.text.NumberFormat` method `setGroupingUsed()` for more information.

The minimum and maximum number of digits in the integer and fractional portions of the formatted output may be given via the `minIntegerDigits`, `maxIntegerDigits`, `minFractionDigits`, and `maxFractionDigits` attributes, respectively. See the `java.text.NumberFormat` methods `setMinimumIntegerDigits()`, `setMaximumIntegerDigits()`, `setMinimumFractionDigits()`, and `setMaximumFractionDigits()` for more information.

Formatting Currencies

When formatting currencies using the specified or locale-dependent formatting pattern for currencies, the currency symbol of the formatting locale is used by default. It can be overridden by using the `currencySymbol` or `currencyCode` attributes, which specify the currency symbol or currency code, respectively, of the currency to use.

If both `currencyCode` and `currencySymbol` are present, `currencyCode` takes precedence over `currencySymbol` if the `java.util.Currency` class is defined in the container's runtime (that is, if the container's runtime is J2SE 1.4 or greater), and `currencySymbol` takes precendence otherwise. If only `currencyCode` is given, it is used as a currency symbol if `java.util.Currency` is not defined.

9.8   <fmt:parseNumber>

Parses the string representation of numbers, currencies, and percentages that were formatted in a locale-sensitive or customized manner.

Syntax

Syntax 1: without a body

```
<fmt:parseNumber   value="numericValue"
                [type="{number|currency|percent}"]
                [pattern="customPattern"]
                [parseLocale="parseLocale"]
                [integerOnly="{true|false}"]
                [var="varName"]
                [scope="{page|request|session|application}"]/>
```

Syntax 2: with a body to specify the numeric value to be parsed

```
<fmt:parseNumber   [type="{number|currency|percent}"]
                [pattern="customPattern"]
                [parseLocale="parseLocale"]
                [integerOnly="{true|false}"]
                [var="varName"]
                [scope="{page|request|session|application}"]>
    numeric value to be parsed
</fmt:parseNumber>
```

Body Content

JSP. The JSP container processes the body content, then the action trims it and processes it further.

Attributes

Name	Dyn	Type	Description
value	true	String	String to be parsed.
type	true	String	Specifies whether the string in the value attribute should be parsed as a number, currency, or percentage.
pattern	true	String	Custom formatting pattern that determines how the string in the value attribute is to be parsed.
parseLocale	true	String or java.util.Locale	Locale whose default formatting pattern (for numbers, currencies, or percentages, respectively) is to be used during the parse operation, or to which the pattern specified via the pattern attribute (if present) is applied.
integerOnly	true	boolean	Specifies whether just the integer portion of the given value should be parsed.
var	false	String	Name of the exported scoped variable which stores the parsed result (of type java.lang.Number).
scope	false	String	Scope of var.

Constraints

- If scope is specified, var must also be specified.

Null & Error Handling

- If the numeric string to be parsed is null or empty, the scoped variable defined by attributes var and scope is removed. This allows "empty" input to be distinguished from "invalid" input, which causes an exception.
- If parseLocale is null or empty, it is treated as if it was missing.
- If an exception occurs during the parsing of the value, it must be caught and rethrown as a JspException. The message of the rethrown JspException must include the value that was to be parsed, and the caught exception must be provided as the root cause.
- If this action fails to determine a formatting locale, it must throw a JspException whose message must include the value that was to be parsed.
- If the attribute pattern is null or empty, it is ignored.

Description

The numeric value to be parsed may be specified via the `value` attribute; if missing, it is read from the action's body content.

The parse pattern may be specified via the `pattern` attribute, or is looked up in a locale-dependent fashion.

A pattern string specified via the `pattern` attribute must follow the pattern syntax specified by `java.text.DecimalFormat`.

If looked up in a locale-dependent fashion, the parse pattern is determined via a combination of the `type` and `parseLocale` attributes. Depending on the value of the `type` attribute, the given numeric value is parsed as a number, currency, or percentage. The parse pattern for numbers, currencies, or percentages is determined by calling the `java.text.NumberFormat` method `getNumberInstance`, `getCurrencyInstance`, or `getPercentInstance`, respectively, with the locale specified via `parseLocale`. If `parseLocale` is missing, the formatting locale, which is obtained according to Section 9.2, is used as the parse locale.

The `pattern` attribute takes precedence over `type`. In either case, the formatting symbols in the pattern (such as decimal separator and grouping separator) are given by the parse locale.

The `integerOnly` attribute specifies whether just the integer portion of the given value should be parsed. See the `java.text.NumberFormat` method `setParseIntegerOnly()` for more information.

If the `var` attribute is given, the parse result (of type `java.lang.Number`) is stored in the named scoped variable. Otherwise, it is output to the current `JspWriter` object using `java.lang.Number.toString()`.

9.9 <fmt:formatDate>

Allows the formatting of dates and times in a locale-sensitive or customized manner.

Syntax

```
<fmt:formatDate value="date"
                [type="{time|date|both}"]
                [dateStyle="{default|short|medium|long|full}"]
                [timeStyle="{default|short|medium|long|full}"]
                [pattern="customPattern"]
                [timeZone="timeZone"]
                [var="varName"]
                [scope="{page|request|session|application}"]/>
```

Body Content

Empty.

Attributes

Name	Dynamic	Type	Description
value	true	java.util. Date	Date and/or time to be formatted.
type	true	String	Specifies whether the time, the date, or both the time and date components of the given date are to be formatted.
dateStyle	true	String	Predefined formatting style for dates. Follows the semantics defined in class java.text.DateFormat. Applied only when formatting a date or both a date and time (i.e. if type is missing or is equal to "date" or "both"); ignored otherwise.
timeStyle	true	String	Predefined formatting style for times. Follows the semantics defined in class java.text.DateFormat. Applied only when formatting a time or both a date and time (i.e. if type is equal to "time" or "both"); ignored otherwise.
pattern	true	String	Custom formatting style for dates and times.

Name	Dynamic	Type	Description
timeZone	true	String or java.util. TimeZone	Time zone in which to represent the formatted time.
var	false	String	Name of the exported scoped variable which stores the formatted result as a `String`.
scope	false	String	Scope of var.

Constraints

- If `scope` is specified, `var` must also be specified.

Null & Error Handling

- If `value` is null, remove the scoped variable if it is specified (see attributes `var` and `scope`).
- If `timeZone` is null or empty, it is handled as if it was missing.
- If this action fails to determine a formatting locale, it uses `java.util.Date.toString()` as the output format.

Description

Depending on the value of the `type` attribute, only the time, the date, or both the time and date components of the date specified via the `value` attribute or the body content are formatted, using one of the predefined formatting styles for dates (specified via the `dateStyle` attribute) and times (specified via the `timeStyle` attribute) of the formatting locale, which is determined according to Section 9.2.

`dateStyle` and `timeStyle` support the semantics defined in `java.text.DateFormat`.

Page authors may also apply a customized formatting style to their times and dates by specifying the `pattern` attribute, in which case the `type`, `dateStyle`, and `timeStyle` attributes are ignored. The specified formatting pattern must use the pattern syntax specified by `java.text.SimpleDateFormat`.

In order to format the current date and time, a <jsp:useBean> action may be used as follows:

```
<jsp:useBean id="now" class="java.util.Date" />
<fmt:formatDate value="${now}" />
```

If the string representation of a date or time needs to be formatted, the string must first be parsed into a `java.util.Date` using the <fmt:parseDate> action, whose parsing result may then be supplied to the <fmt:formatDate> action:

```
<fmt:parseDate value="4/13/02" var="parsed" />
<fmt:formatDate value="${parsed}" />
```

The action's result is output to the current `JspWriter` object, unless the `var` attribute is specified, in which case it is stored in the named scoped variable.

9.10   \<fmt:parseDate\>

Parses the string representation of dates and times that were formatted in a locale-sensitive or customized manner.

Syntax

Syntax 1: without a body
```
<fmt:parseDate   value="dateString"
                [type="{time|date|both}"]
                [dateStyle="{default|short|medium|long|full}"]
                [timeStyle="{default|short|medium|long|full}"]
                [pattern="customPattern"]
                [timeZone="timeZone"]
                [parseLocale="parseLocale"]
                [var="varName"]
                [scope="{page|request|session|application}"]/>
```

Syntax 2: with a body to specify the date value to be parsed
```
<fmt:parseDate   [type="{time|date|both}"]
                [dateStyle="{default|short|medium|long|full}"]
                [timeStyle="{default|short|medium|long|full}"]
                [pattern="customPattern"]
                [timeZone="timeZone"]
                [parseLocale="parseLocale"]
                [var="varName"]
                [scope="{page|request|session|application}"]>
    date value to be parsed
</fmt:parseDate>
```

Body Content

JSP. The JSP container processes the body content, then the action trims it and processes it further.

Attributes

Name	Dyn	Type	Description
value	true	String	Date string to be parsed.
type	true	String	Specifies whether the date string in the value attribute is supposed to contain a time, a date, or both.
dateStyle	true	String	Predefined formatting style for days which determines how the date component of the date string is to be parsed. Applied only when formatting a date or both a date and time (i.e. if type is missing or is equal to "date" or "both"); ignored otherwise.
timeStyle	true	String	Predefined formatting styles for times which determines how the time component in the date string is to be parsed. Applied only when formatting a time or both a date and time (i.e. if type is equal to "time" or "both"); ignored otherwise.
pattern	true	String	Custom formatting pattern which determines how the date string is to be parsed.
timeZone	true	String or java.util.TimeZone	Time zone in which to interpret any time information in the date string.
parseLocale	true	String or java.util.Locale	Locale whose predefined formatting styles for dates and times are to be used during the parse operation, or to which the pattern specified via the pattern attribute (if present) is applied.
var	false	String	Name of the exported scoped variable in which the parsing result (of type java.util.Date) is stored.
scope	false	String	Scope of var.

Constraints

■ If scope is specified, var must also be specified.

Null & Error Handling

- If the date string to be parsed is null or empty, the scoped variable defined by `var` and `scope` is removed. This allows "empty" input to be distinguished from "invalid" input, which causes an exception.
- If `timeZone` is null or empty, it is treated as if it was missing.
- If `parseLocale` is null or empty, it is treated as if it was missing.
- If an exception occurs during the parsing of the value, it must be caught and rethrown as a `JspException`. The message of the rethrown `JspException` must include the value that was to be parsed, and the caught exception must be provided as the root cause.
- If this action fails to determine a formatting locale, it must throw a `JspException` whose message must include the value that was to be parsed.

Description

The date string to be parsed may be specified via the `value` attribute or via the tag's body content.

Depending on the value of the `type` attribute, the given date string is supposed to contain only a time, only a date, or both. It is parsed according to one of the predefined formatting styles for dates (specified via the `dateStyle` attribute) and times (specified via the `timeStyle` attribute) of the locale specified by the `parseLocale` attribute. If the `parseLocale` attribute is missing, the formatting locale, which is determined according to Section 9.2, is used as the parse locale.

If the given date string uses a different format, the pattern required to parse it must be specified via the `pattern` attribute, which must use the pattern syntax specified by `java.text.SimpleDateFormat`. In this case, the `type`, `dateStyle`, and `timeStyle` attributes are ignored. Parsing is non-lenient, i.e. the given date string must strictly adhere to the parsing format.

If the given time information does not specify a time zone, it is interpreted in the time zone determined according to Section 9.4.

If the `var` attribute is given, the parsing result (of type `java.util.Date`) is stored in the named scoped variable. Otherwise, it is output to the current `JspWriter` using the `java.util.Date` method `toString()`.

9.11 Configuration Settings

This section describes the formatting-related configuration settings. Refer to Section 2.8 for more information on how JSTL processes configuration data.

9.11.1 TimeZone

Variable name	`javax.servlet.jsp.jstl.fmt.timeZone`
Java Constant	`Config.FMT_TIMEZONE`
Type	`String` or `java.util.TimeZone`
Set by	`<fmt:setTimeZone>`
Used by	`<fmt:formatDate>`, `<fmt:parseDate>`

Specifies the application's default time zone. A `String` value is interpreted as defined in action <fmt:timeZone> (see Section 9.5).

SQL Actions
sql tag library

Many web applications need to access relational databases as the source of dynamic data for their presentation layer. While it is generally preferred to have database operations handled within the business logic of a web application designed with an MVC architecture, there are situations where page authors require this capability within their JSP pages (e.g. prototyping/testing, small scale/simple applications, lack of developer resources).

The JSTL SQL actions provide the basic capabilities to easily interact with relational databases.

10.1 Overview

The JSTL SQL actions allow page authors to:

- Perform database queries (`select`)
- Easily access query results
- Perform database updates (`insert`, `update`, `delete`)
- Group several database operations into a transaction

10.1.1 Data Source

SQL actions operate on a data source, as defined by the Java class `javax.sql.DataSource`. A `DataSource` object provides connections to the physical data source it represents. Within the context of a `Connection` retrieved from the `DataSource`, SQL statements are executed and results are returned.

A data source can be specified explicitly via the `dataSource` attribute in SQL actions, or it can be totally transparent to a page author by taking advantage of the data source configuration setting (`javax.servlet.jsp.jstl.sql.dataSource`).

There are two ways a data source can be specified as a string.

The first way is through a JNDI relative path, assuming a container supporting JNDI. For example, with the absolute JNDI resource path `java:comp/env/jdbc/myDatabase`, the JNDI relative path to the data source resource would simply be `jdbc/myDatabase`, given that `java:comp/env` is the standard JNDI root for a J2EE application.

The second way is by specifying the parameters needed by the JDBC `DriverManager` class, using the following syntax (see Section 10.6 for details on the JDBC parameters)

```
url[,[driver][,[user][,password]]]
```

For example,

```
jdbc:mysql://localhost/,org.gjt.mm.mysql.Driver
```

where the database has been setup for access without any username or password. If the ',' character occurs in any of the JDBC parameters, it can be escaped by '\'. The character '\' itself can be escaped in the same way.

While the JDBC `DriverManager` class provides a low cost way to use SQL actions, it is not recommended to use it other than for prototyping purposes because it does not provide connection management features one can expect from a properly designed `DataSource` object.

10.1.2   Querying a Database

The most common use of the database actions is to query a database and display the results of the query.

The following sample code selects all customers from China from the customers table in the database, orders them by last name, and finally displays their last name, first name, and address in an HTML table.

```
<sql:query var="customers" dataSource="${dataSource}">
  SELECT * FROM customers
  WHERE country = 'China'
  ORDER BY lastname
</sql:query>

<table>
  <c:forEach var="row" items="${customers.rows}">
    <tr>
      <td><c:out value="${row.lastName}"/></td>
      <td><c:out value="${row.firstName}"/></td>
      <td><c:out value="${row.address}"/></td>
    </tr>
  </c:forEach>
</table>
```

This next example shows a generic way to display the results of a query with column names as headers:

```
<table>
  <!-- column headers -->
  <tr>
    <c:forEach var="columnName" items="${result.columnNames}">
      <th><c:out value="${columnName}"/></th>
    </c:forEach>
  </tr>
  <!-- column data -->
  <c:forEach var="row" items="${result.rowsByIndex}">
    <tr>
      <c:forEach var="column" items="${row}">
        <td><c:out value="${column}"/></td>
      </c:forEach>
    </tr>
  </c:forEach>
</table>
```

10.1.3   Updating a Database

The <sql:update> action updates a database. To ensure database integrity, several updates to a database may be grouped into a transaction by nesting the <sql:update> actions inside a <sql:transaction> action.

For example, the following code transfers money between two accounts in one transaction:

```
<sql:transaction dataSource="${dataSource}">
  <sql:update>
    UPDATE account
    SET Balance = Balance - ?
    WHERE accountNo = ?
    <sql:param value="${transferAmount}"/>
    <sql:param value="${accountFrom}"/>
  </sql:update>
  <sql:update>
    UPDATE account
    SET Balance = Balance + ?
    WHERE accountNo = ?
    <sql:param value="${transferAmount}"/>
    <sql:param value="${accountTo}"/>
  </sql:update>
</sql:transaction>
```

10.1.4   SQL Statement Parameters

The JSTL database actions support substituting parameter values for parameter markers ("?") in SQL statements (as shown in the previous example). This form of parametric replacement is exposed by the SQLExecutionTag interface (see Section 10.10).

The SQLExecutionTag interface is implemented by the tag handlers for <sql:query> and <sql:update>. It is exposed in order to support custom parameter actions. These custom actions may retrieve their parameters from any source and process them before substituting them for a parameter marker in the SQL statement of the enclosing SQLExecutionTag action.

For example, a GUI front end may have a user enter a date as three separate fields (year, month, and day), and use this information in a database query. If the database table being accessed provides only a single column for the complete date, action

<acme:dateParam> could assemble the three separate input parameters into one and pass it to the `addSQLParameter()` method of its enclosing `SQLExecutionTag` action:

```
<sql:update>
  UPDATE PersonalInfo
  SET BirthDate = ?
  WHERE clientId = ?
  <acme:dateParam year="${year}" month="${month}" day="${day}"/>
  <sql:param value="${clientId}"/>
</sql:update>
```

The JSTL formatting tags may be used to parse the string representation of dates and numbers into instances of `java.util.Date` and `java.lang.Number`, respectively, before supplying them to an enclosing `SQLExecutionTag` for parametric replacement:

```
<sql:update sql="${sqlUpdateStmt}" dataSource="${dataSource}">
  <fmt:parseDate var="myDate" value="${someDate}"/>
  <sql:param value="${myDate}"/>
</sql:update>
```

10.2   Database Access

This section describes the algorithm used by the SQL actions (<sql:query>, <sql:update>, <sql:transaction>) to access a database.

- Try to get a reference to a data source as follows:
  - If the attribute `dataSource` is specified, use the value specified for that attribute as the data source.
  - Otherwise, get the configuration setting associated with `javax.servlet.jsp.jstl.sql.dataSource` using `Config.find()` (see Section 2.8). Use the value found as the data source if it is not null.
- If a data source is obtained from the previous step:
  - If it is a `DataSource` object, this is the data source used by the action to access the database.
  - Otherwise, if it is a `String`:
    - Assume this is a JNDI relative path and retrieve the data source from the container's JNDI naming context by concatenating the specified relative path to the J2EE defined root (`java:comp/env/`).
    - If the previous step fails (data source not found), assume the string specifies JDBC parameters using the syntax described in Section 10.1.1 and do as follows:
      - If driver is specified, ensure it is loaded
      - Access the named URL through the `DriverManager` class, using an empty string for `user` or `password` if they are not specified.
    - If the previous step fails, throw an exception.
- Otherwise, throw an exception.

An implementation need not create new objects each time a SQL action is called and the algorithm above does not yield a `DataSource` object directly; i.e when a JNDI path or parameters for the JDBC `DriverManager` class are used. It may reuse objects that it previously created for identical arguments.

10.3   \<sql:query\>

Queries a database.

Syntax

Syntax 1: Without body content
```
<sql:query sql="sqlQuery"
      var="varName" [scope="{page|request|session|application}"]
      [dataSource="dataSource"]
      [maxRows="maxRows"]
      [startRow="startRow"]/>
```

Syntax 2: With a body to specify query arguments
```
<sql:query sql="sqlQuery"
      var="varName" [scope="{page|request|session|application}"]
      [dataSource="dataSource"]
      [maxRows="maxRows"]
      [startRow="startRow"]>

   <sql:param> actions

</sql:query>
```

Syntax 3: With a body to specify query and optional query parameters
```
<sql:query var="varName"
      [scope="{page|request|session|application}"]
      [dataSource="dataSource"]
      [maxRows="maxRows"]
      [startRow="startRow"]>

   query
   optional <sql:param> actions
</sql:query>
```

Body Content

JSP. The JSP container processes the body content, then the action trims it and processes it further.

Attributes

Name	Dynamic	Type	Description
sql	true	String	SQL query statement.
dataSource	true	javax.sql. DataSource or String	Data source associated with the database to query. A String value represents a relative path to a JNDI resource or the parameters for the DriverManager class.
maxRows	true	int	The maximum number of rows to be included in the query result. If not specified, or set to -1, no limit on the maximum number of rows is enforced.
startRow	true	int	The returned Result object includes the rows starting at the specified index. The first row of the original query result set is at index 0. If not specified, rows are included starting from the first row at index 0.
var	false	String	Name of the exported scoped variable for the query result. The type of the scoped variable is javax.servlet.jsp.jstl.sql. Result (see Section 10.9).
scope	false	String	Scope of var.

Constraints

- If dataSource is specified, this action must not be nested inside a
  <sql:transaction>.[1]
- maxRows must be >= -1

Null & Error Handling

- If dataSource is null, a JspException is thrown.
- If an exception occurs during the execution of this action, it must be caught and rethrown as a JspException. The message of the rethrown JspException must include the SQL statement, and the caught exception must be provided as the root cause.

Description

The <sql:query> action queries a database and returns a single result set containing rows of data that it stores in the scoped variable identified by the var and scope attributes.

1. <sql:transaction> is responsible for setting the data source in a transaction.

If the query produces no results, an empty `Result` object (of size zero) is returned.

The SQL query statement may be specified by the `sql` attribute or from the action's body content.

The query statement may contain parameter markers ("?") identifying JDBC `PreparedStatement` parameters, whose values must be supplied by nested parameter actions (such as <sql:param>, see Section 10.7). The <sql:query> action implements the `SQLExecutionTag` interface (see Section 10.10), allowing parameter values to be supplied by custom parameter actions.

maxRows and startRow

The maximum number of rows to be included in the query result may be specified by the `maxRows` attribute (for a specific <sql:query> action) or by the configuration setting `javax.servlet.jsp.jstl.sql.maxRows` (see Section 2.8 and Section 10.12). Attribute `maxRows` has priority over the configuration setting. A value of -1 means that no limit is enforced on the maximum number of rows.

The `startRow` attribute may be used to specify the index of the first row to be included in the returned `Result` object. For example, if given a value of 10, the returned Result object will start with the row located at index 10 (i.e. rows 0 through 9 of the original query result set are skipped). All remaining rows of the original query result set are included.

If both `startRow` and `maxRows` are specified, a maximum of `startRow + maxRows` rows are retrieved from the database. All rows up to `startRow` are then discarded, and the remaining rows (from `startRow` through `startRow + maxRows`) are included in the result.

When using `startRow`, it is important to note that the order in which rows are returned is not guaranteed between RDBMS implementations unless an "order by" clause is specified in the query.

`maxRows` and `startRow` protect against so-called "runaway queries", allow efficient access to the top rows of large result sets, and also provide a "poor-man's way" of paging through a large query result by increasing `startRow` by `maxRows` over a previous page.

Obtaining and Releasing a Connection

If <sql:query> is nested inside an <sql:transaction> action, the `Connection` object is obtained from that parent <sql:transaction> which is reponsible for managing access to the database.

Otherwise, access to the database is performed according to the algorithm described in Section 10.2. A `Connection` object is obtained and released before the action completes.

10.4   <sql:update>

Executes an SQL INSERT, UPDATE, or DELETE statement. In addition, SQL
statements that return nothing, such as SQL DDL statements, can be executed.

Syntax

Syntax 1: Without body content

```
<sql:update sql="sqlUpdate"
    [dataSource="dataSource"]
    [var="varName"] [scope="{page|request|session|application}"]/>
```

Syntax 2: With a body to specify update parameters

```
<sql:update sql="sqlUpdate"
    [dataSource="dataSource"]
    [var="varName"] [scope="{page|request|session|application}"]>
  <sql:param> actions
</sql:update>
```

Syntax 3: With a body to specify update statement and optional update parameters

```
<sql:update [dataSource="dataSource"]
    [var="varName"] [scope="{page|request|session|application}"]>
  update statement
  optional <sql:param> actions
</sql:update>
```

Body Content

JSP. The JSP container processes the body content, then the action trims it and
processes it further.

Attributes

Name	Dyn	Type	Description
`sql`	true	`String`	SQL update statement.
`dataSource`	true	`javax.sql. DataSource` or `String`	Data source associated with the database to update. A String value represents a relative path to a JNDI resource or the parameters for the JDBC `DriverManager` class.
`var`	false	`String`	Name of the exported scoped variable for the result of the database update. The type of the scoped variable is `java.lang.Integer`.
`scope`	false	`String`	Scope of `var`.

Constraints

- If `scope` is specified, `var` must also be specified.
- If `dataSource` is specified, this action must not be nested inside a <sql:transaction>.

Null & Error Handling

- If `dataSource` is null, a `JspException` is thrown.
- If an exception occurs during the execution of this action, it must be caught and rethrown as a `JspException`. The message of the rethrown `JspException` must include the SQL statement, and the caught exception must be provided as the root cause.

Description

The SQL update statement may be specified by the `sql` attribute or from the action's body content.

The update statement may contain parameter markers ("?") identifying JDBC `PreparedStatement` parameters, whose values must be supplied by nested parameter actions (such as <sql:param>, see Section 10.7). The <sql:update> action implements the `SQLExecutionTag` interface (see Section 10.10), allowing the parameter values to be supplied by custom parameter tags.

The connection to the database is obtained in the same manner as described for <sql:query> (see Section 10.3).

The result of an <sql:update> action is stored in a scoped variable named by the `var` attribute, if that attribute was specified. That result represents the number of rows that were affected by the update. Zero is returned if no rows were affected by

INSERT, DELETE, or UPDATE, and for any SQL statement that returns nothing (such as SQL DDL statements). This is consistent with method executeUpdate() of the JDBC class Statement.

10.5   <sql:transaction>

Establishes a transaction context for <sql:query> and <sql:update> subtags.

Syntax

```
<sql:transaction [dataSource="dataSource"]
       [isolation=isolationLevel]>
   <sql:query> and <sql:update> statements
</sql:transaction>

isolationLevel ::= "read_committed"
                 | "read_uncommitted"
                 | "repeatable_read"
                 | "serializable"
```

Body Content

JSP. The JSP container processes the body content and then writes the result to the current `JspWriter`. The action ignores the body content.

Attributes

Name	Dyn	Type	Description
dataSource	true	javax.sql. DataSource or String	DataSource associated with the database to access. A String value represents a relative path to a JNDI resource or the parameters for the JDBC DriverManager facility.
isolation	true	String	Transaction isolation level. If not specified, it is the isolation level the DataSource has been configured with.

Constraints

- Any nested <sql:query> and <sql:update> actions must not specify a `dataSource` attribute.

Null & Error Handling

- If `dataSource` is null, a `JspException` is thrown.

- Any exception occurring during the execution of this action must be caught and rethrown after the transaction has been rolled back (see description below for details).

Description

The <sql:transaction> action groups nested <sql:query> and <sql:update> actions into a transaction.

The transaction isolation levels are defined by `java.sql.Connection`.

The tag handler of the <sql:transaction> action must perform the following steps in its lifecycle methods:

- `doStartTag()`:
  - Determines the transaction isolation level the DBMS is set to (using the `Connection` method `getTransactionIsolation()`).

    If transactions are not supported (that is, the transaction isolation level is equal to `TRANSACTION_NONE`), an exception is raised, causing the transaction to fail.

    For any other transaction isolation level, the auto-commit mode is is saved (so it can later be restored), and then disabled by calling `setAutoCommit(false)`on the `Connection`.
  - If the `isolation` attribute is specified and differs from the connection's current isolation level, the current transaction isolation level is saved (so it can later be restored) and set to the specified level (using the `Connection` method `setTransactionIsolation()`).
- `doEndTag()`: Calls the `Connection` method `commit()`.
- `doCatch()`: Calls the `Connection` method `rollback()`.
- `doFinally()`:
  - If a transaction isolation level has been saved, it is restored using the `Connection` method `setTransactionIsolation()`.
  - Restore auto-commit mode to its saved value by calling `setAutoCommit()` on the `Connection`.
  - Closes the connection.

The `Connection` object is obtained and managed in the same manner as described for <sql:query> (see Section 10.3), except that it is never obtained from a parent tag (<sql:transaction> tags can not be nested as a means to propagate a `Connection`).

Note that the <sql:transaction> tag handler commits or rollbacks the transaction (if it catches an exception) by calling the JDBC `Connection commit()` and `rollback()` methods respectively. Executing the corresponding SQL statements using <sql:update>, e.g. <sql:update sql="rollback" />, within the <sql:transaction> element body is not supported and the result of doing so is unpredictable.

Finally, the behavior of the <sql:transaction> action is undefined if it executes in the context of a larger JTA user transaction.

10.6 <sql:setDataSource>

Exports a data source either as a scoped variable or as the data source configuration
variable (`javax.servlet.jsp.jstl.sql.dataSource`).

Syntax

```
<sql:setDataSource
        {dataSource="dataSource" |
         url="jdbcUrl"
         [driver="driverClassName"]
         [user="userName"]
         [password="password"]}
        [var="varName"]
        [scope="{page|request|session|application}"]/>
```

Body Content

Empty.

Attributes

Name	Dyn	Type	Description
dataSource	true	String or javax.sql.DataSource	Data source. If specified as a string, it can either be a relative path to a JNDI resource, or a JDBC parameters string as defined in Section 10.1.1.
driver	true	String	JDBC parameter: driver class name.
url	true	String	JDBC parameter: URL associated with the database.
user	true	String	JDBC parameter: database user on whose behalf the connection to the database is being made.
password	true	String	JDBC parameter: user password
var	false	String	Name of the exported scoped variable for the data source specified. Type can be `String` or `DataSource`.
scope	false	String	If `var` is specified, scope of the exported variable. Otherwise, scope of the data source configuration variable.

Null & Error Handling

- If `dataSource` is null, a `JspException` is thrown.

Description

If the `var` attribute is specified, the <sql:setDataSource> action exports the data source specified (either as a `DataSource` object or as a String) as a scoped variable. Otherwise, the data source is exported in the `javax.servlet.jsp.jstl.sql.dataSource` configuration variable.

The data source may be specified either via the `dataSource` attribute (as a `DataSource` object, JNDI relative path, or JDBC parameters string), or via the four JDBC parameters attributes. These four attributes are provided as a simpler alternative to the JDBC parameters string syntax defined in Section 10.1.1 that would have to be used with the `dataSource` attribute.

As mentioned in Section 10.1.1, using JDBC's `DriverManager` class to access a database is intended for prototyping purposes only because it does not provide connection management features one can expect from a properly designed `DataSource` object.

10.7   <sql:param>

Sets the values of parameter markers ("?") in a SQL statement. Subtag of
`SQLExecutionTag` actions such as <sql:query> and <sql:update>.

Syntax

Syntax 1: Parameter value specified in attribute "value"
```
<sql:param value="value"/>
```

Syntax 2: Parameter value specified in the body content
```
<sql:param>
    parameter value
</sql:param>
```

Body Content

JSP. The JSP container processes the body content, then the action trims it and
processes it further.

Attributes

Name	Dyn	Type	Description
value	true	Object	Parameter value.

Constraints

- Must be nested inside an action whose tag handler is an instance of
  `SQLExecutionTag` (see Section 10.10).

Null & Error Handling

- If `value` is null, the parameter is set to the SQL value NULL.

Description

The <sql:param> action substitutes the given parameter value for a parameter
marker("?") in the SQL statement of its enclosing `SQLExecutionTag` action.

Parameters are substituted in the order in which they are specified.

The <sql:param> action locates its nearest ancestor that is an instance of `SQLExecutionTag` and calls its `addSQLParameter()` method, supplying it with the given parameter value.

It is important to note that the semantics of `SQLExecutionTag.addSQLParameter()` are such that supplying a parameter with a `String` value (e.g. when using syntax 2) only works for columns of text type (`CHAR`, `VARCHAR` or `LONGVARCHAR`).

10.8   <sql:dateParam>

Sets the values of parameter markers ("?") in a SQL statement for values of type `java.util.Date`. Subtag of `SQLExecutionTag` actions, such as <sql:query> and <sql:update>.

Syntax

```
<sql:dateParam value="value" type="[timestamp|time|date]"/>
```

Body Content

Empty.

Attributes

Name	Dyn	Type	Description
value	true	java.util.Date	Parameter value for DATE, TIME, or TIMESTAMP column in a database table.
type	true	String	One of "date", "time" or "timestamp".

Constraints

- Must be nested inside an action whose tag handler is an instance of `SQLExecutionTag` (see Section 10.10).

Null & Error Handling

- If `value` is null, the parameter is set to the SQL value NULL.

Description

This action converts the provided `java.util.Date` instance to one of `java.sql.Date`, `java.sql.Time` or `java.sql.Timestamp` as defined by the `type` attribute as follows:

- If the `java.util.Date` object provided by the `value` attribute is an instance of `java.sql.Time`, `java.sql.Date`, or `java.sql.Timestamp`, and the `type` attribute matches this object's type, then it is passed as is to the database.

■ Otherwise, the object is converted to the appropriate type by calling that type's constructor with a parameter of `date.getTime()`, where `date` is the value of the `value` attribute.

The <sql:dateParam> action substitutes the given parameter value for a parameter marker("?") in the SQL statement of its enclosing `SQLExecutionTag` action.

Parameters are substituted in the order in which they are specified.

The <sql:dateParam> action locates its nearest ancestor that is an instance of `SQLExecutionTag` and calls its `addSQLParameter()` method, supplying it with the given parameter value.

10.9 Interface Result

This interface represents the result of a <sql:query> action. It provides access to the following information in the query result:

- The result rows (`getRows()` and `getRowsByIndex()`).
- The column names (`getColumnNames()`)
- The number of rows in the result (`getRowCount()`).
- An indication whether the rows returned represent the complete result or just a subset that is limited by a maximum row setting (`isLimitedByMaxRows()`).

Syntax

```
javax.servlet.jsp.jstl.sql
public interface Result
```

Methods

`public java.util.SortedMap[]` **`getRows`**`()`

Returns the result of the query as an array of `SortedMap` objects. Each item of the array represents a specific row in the query result.

A row is structured as a `SortedMap` object where the key is the column name, and where the value is the value associated with the column identified by the key. The column value is an `Object` of the Java type corresponding to the mapping between column types and Java types defined by the JDBC specification when the `ResultSet.getObject()` method is used.

The `SortedMap` must use the `Comparator` `java.util.String.CASE_INSENSITIVE_ORDER`. This makes it possible to access the key as a case insensitive representation of a column name. This method will therefore work regardless of the case of the column name returned by the database.

Returns

The result rows as an array of `SortedMap` objects.

`public Object[][]` **`getRowsByIndex`**`()`

Returns the result of the query as an array of arrays. The first array dimension represents a specific row in the query result. The array elements for each row are `Object` instances of the Java type corresponding to the mapping between column types and Java types defined by the JDBC specification when the `ResultSet.getObject()` method is used.

Returns

The result rows as an array of `Object` arrays.

```
public String[] getColumnNames()
```
   <u>Returns</u>

     The names of the columns in the result. The order of the names in the array
     matches the order in which columns are returned in method
     `getRowsByIndex()`.

```
public int getRowCount()
```
   <u>Returns</u>

     The number of rows in the result.

```
public boolean isLimitedByMaxRows()
```
   <u>Returns</u>

     True if the query was limited by a maximum row setting (i.e. value >= 0).

10.10   Interface SQLExecutionTag

This interface allows tag handlers implementing it to receive values for parameter markers ("?") in their SQL statements.

This interface is implemented by both <sql:query> and <sql:update>. Its `addSQLParameter()` method is called by nested parameter actions (such as <sql:param>) to substitute `PreparedStatement` parameter values for "?" parameter markers in the SQL statement of the enclosing `SQLExecutionTag` action.

Keeping track of the index of the parameter values being added is the responsibility of the tag handler implementing this interface.

The `SQLExecutionTag` interface is exposed in order to support custom parameter actions that may retrieve their parameters from any source and process them before substituting them for a parameter marker in the SQL statement of the enclosing `SQLExecutionTag` action.

Syntax

```
javax.servlet.jsp.jstl.sql
public interface SQLExecutionTag
```

Methods

public void **addSQLParameter**(Object value)

Adds a `PreparedStatement` parameter value. Must behave as if it calls `PreparedStatement.setObject(int, Object)`. For each tag invocation, the integral index passed logically to `setObject()` must begin with 1 and must be incremented by 1 for each subsequent invocation of `addSQLParameter()`. The `Object` logically passed to `setObject()` must be the unmodified object received in the `value` argument.

10.11 Class ResultSupport

Supports the creation of `javax.servlet.jsp.jstl.sql.Result` objects from source `java.sql.ResultSet` objects. `Result` objects make it much easier for page authors to access and manipulate the data resulting from a SQL query.

Syntax

```
javax.servlet.jsp.jstl.sql
public class ResultSupport
```

Methods

`public static Result` **`toResult`**`(java.sql.ResultSet rs)`

Converts a `ResultSet` to a `Result`.

<u>Returns</u>
  The `Result` object.

`public static Result` **`toResult`**`(java.sql.ResultSet rs, int maxRows)`

Converts `maxRows` of a `ResultSet` to a `Result`.

<u>Returns</u>
  The `Result` object.

10.12 Configuration Settings

This section describes the configuration settings used by the SQL actions. Refer to Section 2.8 for more information on how JSTL processes configuration data.

10.12.1 DataSource

Variable name	javax.servlet.jsp.jstl.sql.dataSource
Java Constant	Config.SQL_DATA_SOURCE
Type	String or javax.sql.DataSource
Set by	<sql:setDataSource>, Deployment Descriptor, Config class
Used by	<sql:query>, <sql:update>, <sql:transaction>

The data source to be accessed by the SQL actions. It can be specified as a string representing either a JNDI relative path or a JDBC parameters string (as defined in Section 10.1.1), or as a `javax.sql.DataSource` object.

10.12.2 MaxRows

Variable name	javax.servlet.jsp.jstl.sql.maxRows
Java Constant	Config.SQL_MAX_ROWS
Type	Integer
Set by	Deployment Descriptor, Config class
Used by	<sql:query>

The maximum number of rows to be included in a query result. If the maximum number of rows is not specified, or is -1, it means that no limit is enforced on the maximum number of rows. Value must be >= -1.

XML Core Actions
xml tag library

Enterprise data used in the web tier is increasingly XML these days — when companies cooperate over the web, XML is the data format of choice for exchanging information.

XML is therefore becoming more and more important in a page author's life. The set of XML actions specified in JSTL is meant to address the basic XML needs a page author is likely to encounter.

The XML actions are divided in three categories: XML core actions (this chapter), XML flow control actions (Chapter 12), and XML transform actions (Chapter 13).

11.1 Overview

A key aspect of dealing with XML documents is to be able to easily access their content. XPath, a W3C recommendation since 1999, provides a concise notation for specifying and selecting parts of an XML document. The XML set of actions in JSTL is therefore based on XPath.

The introduction of XPath for the XML tagset expands the notion of expression language. XPath is an expression language that is used locally for the XML actions. Below are the rules of integration that XPath follows as a *local* expression language. These rules ensure that XPath integrates nicely within the JSTL environment.

11.1.1 XPath Context

In XPath, the context for evaluating an expression consists of:

■ A node or nodeset

- Variable bindings (see below)
- Function library

  The default function library comes with the XPath engine. Some engines provide extension functions or allow customization to add new functions. The XPath function library in JSTL is limited to the core function library of the XPath specification.

- Namespace prefix definitions which allow namespace prefixes to be used within an XPath expression.

11.1.2   XPath Variable Bindings

The XPath engine supports the following scopes to easily access web application data within an XPath expression. These scopes are defined in exactly the same way as their implicit object counterparts in the JSTL expression language (see Section A.6).

Expression	Mapping
`$foo`	`pageContext.findAttribute("foo")`
`$param:foo`	`request.getParameter("foo")`
`$header:foo`	`request.getHeader("foo")`
`$cookie:foo`	`maps to the cookie's value for name foo`
`$initParam:foo`	`application.getInitParameter("foo")`
`$pageScope:foo`	`pageContext.getAttribute(` `    "foo", PageContext.PAGE_SCOPE)`
`$requestScope:foo`	`pageContext.getAttribute(` `    "foo", PageContext.REQUEST_SCOPE)`
`$sessionScope:foo`	`pageContext.getAttribute(` `    "foo", PageContext.SESSION_SCOPE)`
`$applicationScope:foo`	`pageContext.getAttribute(` `    "foo", PageContext.APPLICATION_SCOPE)`

Through these mappings, JSP scoped variables, request parameters, headers, and cookies, as well as context init parameters can all be used inside XPath expressions easily. For example:

```
/foo/bar[@x=$param:name]
```

would find the "bar" element with an attribute "x" equal to the value of the http request parameter "name".

11.1.3   Java to XPath Type Mappings

An XPath variable must reference a `java.lang.Object` instance in one of the supported scopes, identified by namespace prefix. The following mappings must be supported:

Java Type	XPath Type
`java.lang.Boolean`	`boolean`
`java.lang.Number`	`number`
`java.lang.String`	`string`
Object exported by <x:parse>	`node-set`

A compliant implementation must allow an XPath variable to address objects exposed by that implementation's handlers for <x:set> and <x:forEach>. For example, while an implementation of <x:set> may expose, for a node-set S, an object of any valid Java type, subsequent XPath evaluations must interpret this object as the node-set S.

An XPath expression must also treat variables that resolve to implementations of standard DOM interfaces as representing nodes of the type bound to that interface by the DOM specification.

XPath variable references that address objects of other types result in implementation-defined behavior. (An implementation may throw an exception if it encounters an unrecognized type.) Following the XPath specification (section 3.1), a variable name that is not bound to any value results in an exception.

11.1.4 XPath to Java Type Mappings

Evaluation of XPath expressions evaluate to XPath types. Their mapping to Java objects is defined as follows:

XPath Type	Java Type
`boolean` true or false	`java.lang.Boolean`
`number` a floating-point number	`java.lang.Number`
`string` a sequence of UCS characters	`java.lang.String`
`node-set` an unordered collection of nodes without duplicates	Type usable by JSTL XML-manipulation tags in the same JSTL implementation. The specific Java type representing node-sets may thus vary by implementation.

11.1.5 The `select` attribute

In all the XML actions of JSTL, XPath expressions are always specified using the `select` attribute. In both the RT and EL versions of the XML tag libraries, `select` is therefore always specified as a string literal that is evaluated by the XPath engine.

This clear separation, where only the `select` attribute of XML actions evaluates XPath expressions, helps avoid confusion between XPath (expression language that is local to the XML actions) and the JSTL expression language (global to all actions with dynamic attributes in the EL version of the tag library).

11.1.6 Resources access

XML actions such as <x:parse> and <x:transform> allow the specification of XML and/or XSLT documents as `String` or `Reader` objects. Accessing a resource through a URL is therefore handled through the <c:import> action that works seamlessly with the XML tags as shown below:

```
<c:import url="http://acme.com/productInfo" var="xml">
  <c:param name="productName" value="${product.name}"/>
</c:import>
<x:parse xml="${xml}" var="doc"/>
```

To resolve references to external entities, the systemId (<**x:parse**>) and xmlSystemId/xsltSystemId (<**x:transform**>) attributes can be used. For these attributes:

- Absolute URLs are passed to the parser directly
- Relative URLs are treated as references to resources (e.g., loaded via ServletContext.getResource()) and loaded using an EntityResolver and URIResolver as necessary

11.1.7   Core Actions

The XML core actions provide "expression language support" for XPath. These actions are therefore similar to the EL support actions <c:out> and <c:set> covered in Chapter 4, except that they apply to XPath expressions.

The core XML actions feature one additional action, <x:parse>, to parse an XML document into a data structure that can then be processed by the XPath engine. For example:

```
<!-- parse an XML document -->
<c:import url="http://acme.com/customer?id=76567" var="xml"/>
<x:parse xml="${xml}" var="doc"/>

<!-- access XML data via XPath expressions -->
<x:out select="$doc/name"/>
<x:out select="$doc/address"/>

<!-- set a scoped variable -->
<x:set var="custName" scope="request" select="$doc/name"/>
```

The context for the evaluation of an XPath Expression can be set either directly within the XPath expression (as shown in the example above), or via an ancestor tag that sets a context that can be used by nested tags. An example of this is with action <**x:forEach**> (see Section 12.6).

```
<!-- context set by ancestor tag <x:forEach> -->
<x:forEach select="$doc//customer">
    <x:out select="name"/>
</x:forEach>
```

11.2   <x:parse>

Parses an XML document.

Syntax

Syntax 1: XML document specified via a String or Reader object

```
<x:parse xml="XMLDocument"
    {var="var" [scope="scope"]|varDom="var" [scopeDom="scope"]}
    [systemId="systemId"]
    [filter="filter"]/>
```

Syntax 2: XML document specified via the body content

```
<x:parse
    {var="var" [scope="scope"]|varDom="var" [scopeDom="scope"]}
    [systemId="systemId"]
    [filter="filter"]>
        XML Document to parse
</x:parse>
```

where *scope* is {<u>page</u>|request|session|application}

Body Content

JSP. The JSP container processes the body content, then the action trims it and
processes it further.

Attributes

Name	Dyn	Type	Description
xml	true	String, Reader	Source XML document to be parsed.
systemId	true	String	The system identifier (URI) for parsing the XML document.
filter	true	org.xml.sax. XMLFilter	Filter to be applied to the source document.
var	false	String	Name of the exported scoped variable for the parsed XML document. The type of the scoped variable is implementation dependent.

Name	Dyn	Type	Description
scope	false	String	Scope for var.
varDom	false	String	Name of the exported scoped variable for the parsed XML document. The type of the scoped variable is org.w3c.dom.Document.
scopeDom	false	String	Scope for varDom.

Null & Error Handling

- If the source XML document is null or empty, a JspException is thrown.
- If filter is null, filtering is not performed.

Description

The <x:parse> action parses an XML document and saves the resulting object in the scoped variable specified by attribute var or varDom. It does not perform any validation against DTDs or Schemas.

The XML document can be specified either with the xml attribute, or inline via the action's body content.

var **and** varDom

If var is used, the type of the resulting object is not defined by this specification. This allows implementations to use whatever they deem best for an efficient implementation of the XML tagset. varDom exposes a DOM document, allowing collaboration with custom actions. Objects exposed by var and varDom can both be used to set the context of an XPath expression.

Filtering for Performance Benefits

If an implementation of the XML tagset is based on DOM-like structures, there will be a significant performance impact when dealing with large XML documents. To help with this, attribute filter can be used to allow filtering of the input data prior to having it parsed by the implementation into a DOM-like structure.

For example, if one is interested in processing only the "European" customers which represent only 10% of the original XML document received as input, it will greatly reduce the size and complexity of the resulting DOM structure if all non-European customers are pruned from the XML document prior to parsing.

```
<c:import url="http://acme.com/customers" var="xml"/>
<x:parse xml="${xml}" filter="${filterEuropeanCust}"
        var="doc"/>
```

The `filter` attribute accepts an object of type `org.xml.sax.XMLFilter`.

If configuration of the filter is desirable, it is suggested that the developer of the filter provides a custom tag for easy configuration by a page author.

11.3   `<x:out>`

Evaluates an XPath expression and outputs the result of the evaluation to the current `JspWriter` object.

Syntax

```
<x:out select="XPathExpression" [escapeXml="{true|false}"]/>
```

Body Content

Empty.

Attributes

Name	Dynamic	Type	Description
select	false	String	XPath expression to be evaluated.
escapeXml	true	boolean	Determines whether characters <,>,&,'," in the resulting string should be converted to their corresponding character entity codes. Default value is true.

Description

The expression to be evaluated is specified via attribute `select` and must be in XPath syntax. The result of the evaluation is converted to a `String` as if the XPath `string()` function were applied, and is subsequently written to the current `JspWriter` object.

This action is the equivalent of `<%=...%>` (display the result of an expression in the JSP syntax) and `<c:out>` (display the result of an expression in the expression language syntax).

If `escapeXml` is true, the following character conversions are applied:

Character	Character Entity Code
<	<
>	>
&	&
'	'
"	"

11.4   <x:set>

Evaluates an XPath expression and stores the result into a scoped variable.

Syntax

```
<x:set select="XPathExpression"
       var="varName" [scope="{page|request|session|application}"]/>
```

Body Content

Empty.

Attributes

Name	Dynamic	Type	Description
select	false	String	XPath expression to be evaluated.
var	false	String	Name of the exported scoped variable to hold the *value* specified in the action. The type of the scoped variable is whatever type the select expression evaluates to.
scope	false	String	Scope for var.

Description

Evaluates an XPath expression (specified via attribute select) and stores the result into a scoped variable (specified via attributes var and scope).

The mapping of XPath types to Java types is described in Section 11.1.4.

XML Flow Control Actions
xml tag library

The core set of XML actions provides the basic functionality to easily parse and access XML data. Another important piece of functionality is the ability to iterate over elements in an XML document, as well as conditionally process JSP code fragments depending on the result of an XPath expression. The XML flow control actions provide these capabilities.

12.1 Overview

The XML flow control actions provide flow control based on the value of XPath expressions. These actions are therefore similar to the EL flow control actions (<c:if>, <c:choose>, and <c:forEach>), except that they apply to XPath expressions.

The <x:if> action has a `select` attribute that specifies an XPath expression. The expression is evaluated and the resulting object is converted to a `boolean` according to the semantics of the XPath `boolean()` function:

- A number is true if an only if it is neither positive or negative zero nor NaN
- A node-set is true if and only if it is non-empty
- A string is true if and only if its length is non-zero

<x:if> renders its body if the result is true. For example:

```
<x:if select="$customer/[location='UK']">
   UK based customer
</x:if>
```

The <x:choose> action selects one among a number of possible alternatives. It consists of a sequence of <x:when> elements followed by an optional <x:otherwise>. Each <x:when> element has a single attribute, `select`, which specifies an XPath expression. When a <x:choose> element is processed, each of the <x:when> elements has its expression evaluated in turn, and the resulting object is converted to a boolean according to the semantics of the XPath boolean function. The body of the first, and only the first, <x:when> whose result is true is rendered.

If none of the test conditions of nested <x:when> tags evaluates to true, then the body of an <x:otherwise> tag is evaluated, if present.

```
<x:choose>
    <x:when select="$customer/firstName">
        Hello <x:out select="$customer/firstName"/>
    </x:when>
    <x:otherwise>
        Hello my friend
    </x:otherwise>
</x:choose>
```

The <x:forEach> action evaluates the given XPath expression and iterates over the result, setting the context node to each element in the iteration. For example:

```
<x:forEach select="$doc//author">
    <x:out select="@name"/>
</x:forEach>
```

12.2   \<x:if\>

Evaluates the XPath expression specified in the `select` attribute and renders its body content if the expression evaluates to true.

Syntax

Syntax 1: Without body content

```
<x:if select="XPathExpression"
      var="varName" [scope="{page|request|session|application}"]/>
```

Syntax 2: With body content

```
<x:if select="XPathExpression"
      [var="varName"] [scope="{page|request|session|application}"]>

   body content

</x:if>
```

Body Content

JSP. If the test condition evaluates to true, the JSP container processes the body content and then writes it to the current `JspWriter`.

Attributes

Name	Dynamic	Type	Description
select	false	String	The test condition that tells whether or not the body content should be processed.
var	false	String	Name of the exported scoped variable for the resulting value of the test condition. The type of the scoped variable is `Boolean`.
scope	false	String	Scope for `var`.

Constraints

■ If `scope` is specified, `var` must also be specified.

Description

The XPath expression specified via attribute `select` is evaluated, and the resulting object is converted to a `boolean` according to the semantics of the XPath `boolean()` function. If true, the body content is evaluated by the JSP container and the result is written to the current `JspWriter`.

12.3 <x:choose>

Provides the context for mutually exclusive conditional execution.

Syntax

```
<x:choose>
   body content (<x:when> and <x:otherwise> subtags)
</x:choose>
```

Body Content

JSP. The body content is processed by the JSP container (at most one of the nested elements will be processed) and written to the current `JspWriter`.

Constraints

- The body of the <x:choose> action can only contain:
  - White spaces

    May appear anywhere around the <x:when> and <x:otherwise> subtags.
  - 1 or more <x:when> actions

    Must all appear before <x:otherwise>
  - 0 or 1 <x:otherwise> action

    Must be the last action nested within <x:choose>

Description

The <x:choose> action processes the body of the first `<x:when>` action whose test condition evaluates to true. If none of the test conditions of nested `<x:when>` actions evaluates to true, then the body of an `<x:otherwise>` action is processed, if present.

12.4   <x:when>

Represents an alternative within an <x:choose> action.

Syntax

```
<x:when select="XPathExpression">
   body content
</x:when>
```

Body Content

JSP. If this is the first `<x:when>` action to evaluate to true within `<x:choose>`, the JSP container processes the body content and then writes it to the current `JspWriter`.

Attributes

Name	Dynamic	Type	Description
select	false	String	The test condition that tells whether or not the body content should be processed

Constraints

- Must have `<x:choose>` as an immediate parent.
- Must appear before an `<x:otherwise>` action that has the same parent.

Description

The XPath expression specified via attribute `select` is evaluated, and the resulting object is converted to a `boolean` according to the semantics of the XPath `boolean()` function. If this is the first `<x:when>` action to evaluate to true within `<x:choose>`, the JSP container processes the body content and then writes it to the current `JspWriter`.

12.5   <x:otherwise>

Represents the last alternative within a <x:choose> action.

Syntax

```
<x:otherwise>
    conditional block
</x:otherwise>
```

Body Content

JSP. If no <x:when> action nested within <x:choose> evaluates to true, the JSP container processes the body content and then writes it to the current `JspWriter`.

Attributes

None.

Constraints

- Must have <x:choose> as an immediate parent.
- Must be the last nested action within <x:choose>.

Description

Within a <x:choose> action, if none of the nested <x:when> test conditions evaluates to true, then the body content of the <x:otherwise> action is evaluated by the JSP container, and the result is written to the current `JspWriter`.

12.6   <x:forEach>

Evaluates the given XPath expression and repeats its nested body content over the result, setting the context node to each element in the iteration.

Syntax

```
<x:forEach[var="varName"] select="XPathExpression">
    body content
</x:forEach>
```

Body Content

JSP. As long as there are items to iterate over, the body content is processed by the JSP container and written to the current `JspWriter`.

Attributes

Name	Dynamic	Type	Description
var	false	String	Name of the exported scoped variable for the current item of the iteration. This scoped variable has nested visibility. Its type depends on the result of the XPath expression in the `select` attribute.
select	false	String	XPath expression to be evaluated.

Null & Error Handling

■ If `select` is empty, a `JspException` is thrown.

XML Transform Actions
xml tag library

The transformation of XML documents using XSLT stylesheets is popular in many web applications. The XML transform actions provide this capability so XSLT transformations can be performed within JSP pages.

13.1   Overview

The XML transform actions support the transformation of XML documents with XSLT stylesheets.

In the example below, an external XML document (retrieved from an absolute URL) is transformed by a local XSLT stylesheet (context relative path). The result of the transformation is written to the page.

```
<c:import url="http://acme.com/customers" var="xml"/>
<c:import url="/WEB-INF/xslt/customerList.xsl" var="xslt"/>
<x:transform xml="${xml}" xslt="${xslt}"/>
```

It is possible to set transformation parameters via nested <x:param> actions. For example:

```
<x:transform xml="${xml}" xslt="${xslt}">
    <x:param name="foo" value="foo-value"/>
</x:transform>
```

It is sometimes the case that the same stylesheet transformation needs to be applied multiple times to different source XML documents. A more efficient approach is to process the transformation stylesheet once, and then save this "transformer" object for successive transformations. The specification allows implementations to support transparent caching of transformer objects to improve performance.

13.2 <x:transform>

Applies an XSLT stylesheet transformation to an XML document.

Syntax

Syntax 1: Without body contentt
```
<x:transform
      xml="XMLDocument" xslt="XSLTStylesheet"
      [xmlSystemId="XMLSystemId"] [xsltSystemId="XSLTSystemId"]
      [{var="varName" [scope="scopeName"]|result="resultObject"}]
```

Syntax 2: With a body to specify transformation parameters
```
<x:transform
      xml="XMLDocument" xslt="XSLTStylesheet"
      [xmlSystemId="XMLSystemId"] [xsltSystemId="XSLTSystemId"]
      [{var="varName" [scope="scopeName"]|result="resultObject"}]
   <x:param> actions
</x:transform>
```

Syntax 3: With a body to specify XML document and optional transformation parameters
```
<x:transform
      xslt="XSLTStylesheet"
      xmlSystemId="XMLSystemId" xsltSystemId="XSLTSystemId"
      [{var="varName" [scope="scopeName"]|result="resultObject"}]
   XML Document to parse
   optional <x:param> actions
</x:parse>
```

where `scopeName` is {<u>page</u>|request|session|application}

Body Content

JSP. The JSP container processes the body content, then the action trims it and processes it further.

Attributes

Name	Dyn	Type	Description
xml	true	String, Reader, javax.xml.transform.Source, org.w3c.dom.Document, or object exported by <x:parse>, <x:set>.	Source XML document to be transformed. (If exported by <x:set>, it must correspond to a well-formed XML document, not a partial document.)
xslt	true	String, Reader or javax.xml.transform.Source	Transformation stylesheet as a String, Reader, or Source object.
xmlSystemId	true	String	The system identifier (URI) for parsing the XML document.
xsltSystemId	true	String	The system identifier (URI) for parsing the XSLT stylesheet.
var	false	String	Name of the exported scoped variable for the transformed XML document. The type of the scoped variable is org.w3c.dom.Document.
scope	false	String	Scope for var.
result	true	javax.xml.transform.Result	Object that captures or processes the transformation result.

Null & Error Handling

- If the source XML document is null or empty, a JspException is thrown.
- If the source XSLT document is null or empty, a JspException is thrown.

Description

The <x:transform> tag applies a transformation to an XML document (attribute xml or the action's body content), given a specific XSLT stylesheet (attribute xslt). It does not perform any validation against DTD's or Schemas.

Nothing prevents an implementation from caching Transformer objects across invocations of <x:transform>, though implementations should be careful they take into account both the xslt and xsltSystemId attributes when deciding whether to

use a cached `Transformer` or produce a new one. An implementation may assume that any external entities that were referenced during parsing will not change values during the life of the application.

The result of the transformation is written to the page by default. It is also possible to capture the result of the transformation in two other ways:

- `javax.xml.transform.Result` object specified by the `result` attribute.
- `org.w3c.dom.Document` object saved in the scoped variable specified by the `var` and `scope` attributes.

13.3   <x:param>

Set transformation parameters. Nested action of <x:transform>.

Syntax

Syntax 1: Parameter value specified in attribute "value"

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

Syntax 2: Parameter value specified in the body content

```
<x:param name="name">
    parameter value
</x:param>
```

Body Content

JSP. The JSP container processes the body content, then the action trims it and processes it further.

Attributes

Name	Dynamic	Type	Description
name	true	String	Name of the transformation parameter.
value	true	Object	Value of the parameter.

Description

The <x:param> action must be nested within <x:transform> to set transformation parameters. The value of the parameter can be specified either via the value attribute, or via the action's body content.

Tag Library Validators

JSP 1.2 provides tag library validators (TLVs) as a mechanism for a tag library to enforce constraints on the JSP document (the "XML view") associated with any JSP page into which the tag library is imported. While the expectation is that TLVs used by a tag library will typically enforce multi-tag constraints related to usage of the library's tags themselves, a TLV is free to perform arbitrary validation of JSP documents. A TLV returns to the container information about which elements, if any, are in violation of its specific constraints, along with textual descriptions of the syntactic violation.

JSTL provides TLVs that perform "reusable" validation; i.e. generic validation that custom tag-library authors might wish to incorporate in their own tag libraries. These tag libraries do not necessarily need to be substantial collections of tags; a taglib may exist simply to provide site-specific validation logic. Just like tag libraries whose primary focus is to provide new tags, such "validation-centric" tag libraries may be configured and used by "back-end" developers in order to affect the "front-end" JSP page author's environment.

This chapter covers the JSTL tag library validators.

14.1   Overview

JSTL exposes via TLVs two simple types of validations. These TLV classes may be used in custom tag-library descriptors (TLDs) to restrict the page author's activities. The two types of validation provided in this fashion are:

- Assurance of script-free pages
- Enumeration of permitted tag libraries (including JSTL) on a page

For example, to prevent a JSP page from using JSP scriptlets and JSP declarations, but still allow expressions, a developer could create the following TLD:

```
<taglib>
    <tlib-version>1.0</tlib-version>
    <jsp-version>1.2</jsp.version>
    <short-name>scriptfree</scriptfree>
    <uri>http://acme.com/scriptfree</uri>
    <description>
        Validates JSP pages to prohibit use of scripting elements.
    </description>

    <validator>
        <validator-class>
            javax.servlet.jsp.jstl.tlv.ScriptFreeTLV
        </validator-class>
        <init-param>
            <param-name>allowDeclarations</param-name>
            <param-value>false</param-value>
        </init-param>
        <init-param>
            <param-name>allowScriptlets</param-name>
            <param-value>false</param-value>
        </init-param>
        <init-param>
            <param-name>allowExpressions</param-name>
            <param-value>true</param-value>
        </init-param>
        <init-param>
            <param-name>allowRTExpressions</param-name>
            <param-value>true</param-value>
        </init-param>

    </validator>

    <tag>
        <!-- Need at least one no-op tag because the TLD DTD
             requires at least one <tag> element -->
        ...
    </tag>
</taglib>
```

Similarly, to restrict a JSP page to a set of permitted tag-libraries (in the example below, the JSTL "EL" tag libraries), a developer could create the following TLD:

```
<taglib>
    <tlib-version>1.0</tlib-version>
    <jsp-version>1.2</jsp.version>
    <short-name>jstl taglibs only</scriptfree>
    <uri>http://acme.com/jstlTaglibsOnly</uri>
    <description>
        Restricts JSP pages to the JSTL tag libraries
    </description>

    <validator>
        <validator-class>
            javax.servlet.jsp.jstl.tlv.PermittedTaglibsTLV
        </validator-class>
        <init-param>
            <param-name>permittedTaglibs</param-name>
            <param-value>
                http://java.sun.com/jstl/core
                http://java.sun.com/jstl/xml
                http://java.sun.com/jstl/fmt
                http://java.sun.com/jstl/sql
            </param-value>
        </init-param>
    </validator>

    <tag>
        <!-- Need at least one no-op tag because the TLD DTD
             requires at least one <tag> element -->
        ...
    </tag>
</taglib>
```

14.2 Class ScriptFreeTLV

The ScriptFree tag library validator enforces restrictions against the use of JSP scripting elements.

Class

```
javax.servlet.jsp.jstl.tlv.ScriptFreeTLV
```

Initialization Parameters

There are four initialization parameters that control which of the four types of scripting elements are allowed or prohibited.

The default value for all for initialization parameters is `false`, indicating all forms of scripting elements are to be prohibited.

allowDeclarations

  Indicates whether declaration elements are prohibited or not. Default value is `false`.

allowScriptlets

  Indicates whether scriptlets are prohibited or not. Default value is `false`.

allowExpressions

  Indicates whether top-level expression elements (i.e., expressions not associated with request-time attribute values) are prohibited or not. Default value is `false`.

allowRTExpressions

  Indicates whether expression elements associated with request-time attribute values are prohibited or not. Default value is `false`.

14.3   Class PermittedTaglibsTLV

The PermittedTaglibs tag library validator enforces restrictions against the set of tag libraries that are allowed in a JSP page.

Class

javax.servlet.jsp.jstl.tlv.PermittedTaglibsTLV

Initialization Parameters

permittedTaglibs

A whitespace-separated list of URIs corresponding to tag libraries permitted to be imported on the page in addition to the tag library that references PermittedTaglibsTLV (which is allowed implicitly).

Expression Language Definition

This document describes an expression language (EL) for JSP. This expression language has two versions: one applicable to the JSP Standard Tag Library (JSTL), the other applicable to JSP 1.3. Both expert groups have collaborated to design the expression language, but their schedules are different, and the requirements vary somewhat too.

This document describes the expression language for JSTL 1.0. Ideally, this version of the expression language will be compatible with that used in JSP 1.3, but we can't rule out that there may be changes that are deemed necessary by the expert groups. If that turns out to be the case, we will incorporate those changes into a maintenance release of JSTL.

A.1 Overview

The EL is a simple language based on:

- Avariable namespace (the `PageContext` attributes)
- Nested properties and accessors to collections
- Relational, logical and arithmetic operators.
- A set of implicit objects

The EL is inspired by both ECMAScript and the XPath expression languages. The expert groups were very reluctant to design yet another expression language and tried to use each of these languages but fell short in different areas.

A.2 Expressions and Attribute Values

In JSTL, the EL is available only in attribute values (in JSP 1.3, the EL can also be used within template text). The EL is invoked exclusively via the construct `${expr}`.

There are three use cases for expressions in attribute values:

- The attribute value contains a single expression construct

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

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

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

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

  In this case, the expressions are evaluated from left to right, coerced to Strings (according to the type conversion rules described later), and concatenated with any intervening text. The resulting `String` is then coerced to the attribute's expected type according to the type conversion rules described later.

- The attribute value contains only text:

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

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

A.2.1 Examples

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

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

Note that the normal JSP coercion mechanism already allows for:

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

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

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

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

A.3   Syntax

A.3.1   Overview

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

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

A.3.2   Literals

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

- Boolean - `true` and `false`
- Integer - As defined by the `IntegerLiteral` construct in Section A.8
- Floating point - As defined by the `FloatingPointLiteral` construct in Section A.8
- String - With single and double quotes - " is escaped as \", ' is escaped as \', and \ is escaped as \\ . Quotes only need to be escaped in a string value enclosed in the same type of quote
- Null - `null`

A.3.3   Errors, Warnings, Default Values

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

A.3.4   Operators "[]" and "."

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

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

To evaluate `expr-a[expr-b]`:

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

A.3.5   Arithmetic Operators

Arithmetic is provided to act on integer (long) and floating point values. There are 5 operators:

- Addition: "+"
- Substraction: "-"
- Multiplication: "*"
- Division: "/" and "div"
- Remainder (modulo): "%" and "mod"

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

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

A.3.5.1   Binary operators - A {+,-,*} B

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

A.3.5.2   Binary operator - A {/} B

- If A and B are null, - return 0
- Coerce both A and B to `Double` and apply operator
- If operator results in exception, error

A.3.5.3   Binary operator - A {%} B

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

A.3.5.4   Unary minus operator - -A

- If A is null, return 0
- If A is a `String`
  - If A contains ".", "e", or "E", coerce to a `Double` and apply operator

- Otherwise, coerce to a `Long` and apply operator
- If operator results in exception, error
- If A is `Byte, Short, Integer, Long, Float, Double`
  - Retain type, apply operator
  - If operator results in exception, error
- Otherwise, error

A.3.6   Relational Operators

The relational operators are:

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

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

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

A.3.6.1   A {<,>,<=,>=,lt,gt,le,ge} B

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

A.3.6.2   A {==,!=,eq,ne} B

- If A==B, apply operator
- If A is null or B is null return `false` for ==, `true` for !=
- If A or B is `Float` or `Double` coerce both A and B to `Double` apply operator

- If A or B is `Byte`, `Short`, `Character`, `Integer`, `Long` coerce both A and B to `Long` apply operator
- If A or B is `Boolean` coerce both A and B to `Boolean` apply operator
- If A or B is `String` coerce both A and B to `String`, compare lexically
- Otherwise if an error occurs while calling A.equals(B), error
- Otherwise, apply operator to result of A.equals(B)

A.3.7   Logical Operators

The logical operators are:

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

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

A.3.7.1   Binary operator - A {&&,||,and,or} B

- Coerce both A and B to `Boolean`, apply operator

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

A.3.7.2   Unary not operator - {!,not} A

- Coerce A to `Boolean`, apply operator

A.3.8   Empty Operator - empty A

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

To evaluate empty A

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

> **Note –** Other types may be added in future revisions of the specification.

A.3.9 Others

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

A.3.10 Operator Precedence

Highest to lowest, left-to-right.

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

A.4 Reserved Words

The following words are reserved for the language and should not be used as identifiers without being quoted.

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

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

A.5   Named Variables

The EL evaluates an identifier by looking up its value as an attribute, according to the behavior of `PageContext.findAttribute(String)`. For example:

```
${product}
```

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

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

A.6   Implicit Objects

The EL defines a set of implicit objects. When an expression references one of these objects by name, the appropriate object is returned instead of the corresponding attribute. For example:

```
${pageContext}
```

returns the `PageContext` object, even if there is an existing `pageContext` attribute containing some other value.

The following implicit objects are available:

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

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

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

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

A.7 Type Conversion

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

A.7.1 Coerce A to String

- If A is `String`: return A
- Otherwise, if A is null: return "".
- Otherwise, if `A.toString()` throws exception, error
- Otherwise, return `A.toString()`
- Otherwise, if A is a primitive type, return a `String` as in the java expression `""+A`

A.7.2   Coerce A to primitive Number type N

- If A is null or "", return 0.
- If A is `Character`, convert to `short`, apply following rules.
- If A is `Boolean`, then error.
- If A is `Number` type N, return A
- If A is `Number` with less precision than N, coerce quietly
- If A is `Number` with greater precision than N, coerce quietly
- If A is `String`, and new `N.valueOf(A)` throws exception, then error.
- If A is `String`, and `N.valueOf(A)` does not throw exception, return it.
- Otherwise, error.

A.7.3   Coerce A to Character

- If A is null or "", return `(char) 0`
- If A is `Character`, return A
- If A is `Boolean`, error
- If A is `Number` with less precision than short, coerce quietly – return `(char) A`
- If A is `Number` with greater precision than short, coerce quietly – return `(char) A`
- If A is `String`, return `A.charAt (0)`
- Otherwise, error

A.7.4   Coerce A to Boolean

- If A is null or "", return false
- Otherwise, if A is `Boolean`, return A
- Otherwise, if A is `String`, and `Boolean.valueOf(A)` throws exception, error.
- Otherwise. if A is `String` return `Boolean.valueOf(A)`
- Otherwise, error

A.7.5   Coerce A to any other type T

- If A is null, return null
- If A is assignable to T, coerce quietly
- If A is a primitive type, return a boxed object with equivalent value, as follows:
  - If A is an `int`, return `new Integer(A)`
  - If A is a `float`, return `new Float(A)`
- If A is `String`, and T has no `PropertyEditor`
  - If A is "", return null
  - Otherwise error
- If A is `String` and T's `PropertyEditor` throws exception

- If A is "", return null
- Otherwise, error
- Otherwise, apply T's `PropertyEditor`.
- Otherwise, error

A.8   Syntax

The following are the constructs supported by the EL:

Expression	::=-	**Expression BinaryOp Expression** **| UnaryExpression**
BinaryOp	::=-	'and' | '&&' | 'or' | '||' | '+' | '-' | '*' | '/' | 'div' | '%' | 'mod' | '>' | 'gt' | '<' | 'lt' | '>=' | 'ge' | '<=' | 'le' | '==' | 'eq' | '!=' | 'ne'
UnaryExpression	::=-	**UnaryOp UnaryExpression** **| Value**
UnaryOp	::=-	'-' | 'not'

Value	::=-	**ValuePrefix**
		| **Value ValueSuffix**
ValuePrefix	::=-	**Literal**
		| '(' **Expression** ')'
		| **Identifier** except for **ImplicitObject**
		| **ImplicitObject**
ValueSuffix	::=-	'.'**Identifier**
		| [**Expression**]
Identifier	::=-	*Java language identifier*
ImplicitObject	::=-	'pageContext'
		| 'pageScope'
		| 'requestScope'
		| 'sessionScope'
		| 'applicationScope'
		| 'param'
		| 'paramValues'
		| 'header'
		| 'headerValues'
		| 'initParam'
		| 'cookie'
Literal	::=-	**BooleanLiteral**
		| **IntegerLiteral**
		| **FloatingPointLiteral**
		| **StringLiteral**
		| **NullLiteral**
BooleanLiteral	::=-	'true'
		| 'false'

StringLiteral ::=- '([^'\]|\'|\\)*'
| "([^'\]|\'|\\)*"

i.e., a string of any characters enclosed by single or double quotes, where \ is used to escape ', ", and \. It is possible to use single quotes within double quotes, and vice versa, without escaping.

IntegerLiteral ::=- ["+","-"] ["1"-"9"] (["0"-"9"])*

FloatingPointLiteral ::=- (["+","-"])? (["0"-"9"])+ "." (["0"-"9"])* (["e","E"] (["+","-"])? (["0"-"9"])+)?
| (["+","-"])? "." (["0"-"9"])+ (["e","E"] (["+","-"])? (["0"-"9"])+)?
| (["+","-"])? (["0"-"9"])+ ["e","E"] (["+","-"])? (["0"-"9"])+

NullLiteral ::=- 'null'

Notes:

- An identifier is constrained to be a Java identifier - i.e., no "-", no "/", etc. without quoting.
- A `String` only recognizes a limited set of escape sequences, and \ may not appear unescaped. This may eventually be expanded, but once again, this is the minimum.
- The definitions for integer and floating point literals are pretty much taken from the Java definitions.
- The relational operator for equality is "==" (double equals).

Changes

This appendix lists the changes in the JSTL 1.0 specification. This appendix is non-normative.

B.1 Changes between Proposed Final Draft and Final Draft

Many typos and clarifications have been made to the specification. Clarifications and modifications worth noting include:

Preface

- Added typographical conventions.

Chapter 2 - **Conventions**

- When an action is required to throw an exception, there were two choices when no root cause was involved: `JspException` or `JspTagException`. The specification has now standardized on `JspException` everywhere in the spec (instead of `JspException` in some places (with root cause), and `JspTagException` in some others (no root cause)).
- Clarified the proper handling of constraints in section 2.7.
- Constants names now use "_" as word separators (e.g. FMT_FALLBACK_LOCALE)

Chapter 3 - **Expression Language Overview**

- Fixed example featuring the `default` attribute in section 3.6.

Chapter 4 - **General-Purpose Actions**

- Transparent conversion now supported on a value to be set as a bean property.

- Clarified behavior of <c:set> when value is null, so it has the same semantics as <c:remove>.

- Clarified the behavior of <c:out> when value is null.

Chapter 6 - Iterator Actions

- Corrected the name of method setStatus() to be setVarStatus(), as it should have been.

- Methods next(), hasNext(), prepare() of class LoopTagSupport are abstract methods.

- Method hasNext()of class LoopTagSupport returns boolean.

- Added protected fields beginSpecified, endSpecified, and stepSpecified to class LoopTagSupport.

Chapter 8 - I18N Actions

- Left over references to javax.servlet.jsp.jstl.fmt.bundle have been changed to javax.servlet.jsp.jstl.fmt.localizationContext.

- Added the three constructors to class LocalizationContext and clarified the behavior of methods getResourceBundle() and getLocale().

Chapter 9 - Formatting Actions

- Clarified how the formatting pattern applies in <fmt:number> and <fmt:parseNumber>.

Chapter 10 - SQL Actions

- Clarified the handling of auto-commit mode and isolation level in <sql:transaction>.

- Clarified the handling of exceptions occurring during the execution of <sql:transaction>.

- Added clarification to <sql:param> when dealing with String values (only works for columns of text type).

- Clarified that if dataSource is null, a JspException is thrown for <sql:query>, <sql:update>, <sql:transaction>, and <sql:setDataSource>.

Chapters 11, 12, 13 - XML Actions

- Clarified "Null & Error Handling" for <x:parse> and <x:transform>

- In <x:forEach>, if select is empty, a JspException is now thrown.

- Added syntax without body content to <x:if>. It is now similar to <c:if>.

- Only String and Reader objects are now allowed for the xml attribute of <x:parse>.

- Clarified that DOM objects are supported as XPath variables.

Appendix A - Expression Language

- Alternative operators &&, ||, and ! were missing in some of the tables. They now appear along with their counterpart and, or, and not.
- Clarified the definition of integer and floating point literals.
- Removed division by 0 as an example of exception for arithmetic operators / and %.

B.2 Changes between Public Draft and Proposed Final Draft

Many typos and clarifications have been made to the specification. Major changes include:

Preface

- Added acknowledgements.

Chapter 1 - **Introduction**

- Clarified the fact that actions from EL- and RT- based libraries can be mixed together.

Chapter 2 - **Conventions**

- Clarified how actions throw exceptions.
- "Section 2.8 - Configuration Parameters" has been completely rewritten and is now titled "Configuration Data". The way configuration data is handled in JSTL has been clarified and will now work properly with containers that implement JSP scopes via a single namespace.

Chapter 4 - **Expression Language Support Actions**

- Renamed the chapter to "General Purpose Actions".
- Removed the restriction that the actions in this chapter are only available in the EL-based version of the library.
- Extended the scope of <c:set> so it supports setting a property of a target JavaBeans or `java.util.Map` object.

Chapter 7 - **URL Related Actions**

- Improved the error handling behavior of <c:import>
- <c:url> and <c:redirect> now append the context path to any relative URL that starts with "/". Added new attribute `context` to properly handle foreign context URLs.

Chapter 8 - **I18N Actions**

- In the resource bundle lookup, the locale-less root resource bundle is now supported if neither the preferred locales nor the fallback locale yield a resource bundle match.

- <fmt:locale> has been renamed to <fmt:setLocale>.

- <fmt:bundle> no longer takes 'var' and 'scope'. Creating and storing an I18N localization context with a resource bundle in a 'var' or scoped configuration variable is now done by the new <fmt:setBundle>.

- Logging is considered an implementation-specific (deployment) issue and has been removed from <fmt:message>'s description.

- A new class `LocalizationContext` has been defined which represents an I18N localization context containing a `java.util.ResourceBundle` and a `java.util.Locale`.

- `javax.servlet.jsp.jstl.fmt.basename` has been replaced with `javax.servlet.jsp.jstl.fmt.localizationContext`.

Chapter 9 - Formatting Actions

- Formatting actions nested inside a <fmt:bundle> no longer use that bundle's locale as their formatting locale, but the locale of the enclosing I18N localization context, which is the (possibly more specific) locale that led to the resource bundle match.

- <fmt:timeZone> no longer takes 'var' and 'scope'. Storing a time zone in a 'var' or scoped configuration variable is now done by the new <fmt:setTimeZone>.

- <fmt:formatNumber> no longer uses the "en" locale to parse numeric values given as strings, but uses Long.valueOf() or Double.valueOf() instead.

- In <fmt:parseNumber>, `parseLocale`, which used to support string values only, now also supports values of type `java.util.Locale`.

- <fmt:formatDate> no longer supports literal values, and no longer has a body. Its 'value' attribute is no longer optional, meaning the default behaviour of formatting the current time and date is no longer supported.

- In <fmt:parseDate>, `parseLocale`, which used to support string values only, now also supports values of type `java.util.Locale`.

- <fmt:setLocale>, formerly known as <fmt:locale>, now also accepts values of type `java.util.Locale` (in addition to string values).

- The runtime's default locale is no longer used as a fallback, since it is not guaranteed to be among the supported formatting locales.

- <fmt:timeZone> and the new <fmt:setTimeZone> now also accept values of type `java.util.TimeZone` (in addition to string values).

Chapter 10 - SQL Actions

- The configuration settings now include JDBC parameters.

- <sql:driver> has been renamed <sql:setDataSource>. It now supports attribute "password" as well as setting configuration variables.

- The keys in the Map objects returned by Result.getRows() are now case-insensitive. The motivation for this change is that some databases return column names as all-uppercase strings in the ResultSet, while others return them with the same upper/lowercase mix as was used in the SELECT statement.

- Method `Result.getRowsCount()` has been renamed to `Result.getRowCount()` to be compatible with naming conventions in J2SE.

- Method `Result.getMetaData()` as well as interface `ColumnMetaData` have been removed because handling of exceptions encountered when caching `ResultSetMetaData` is problematic. New method `Result.getColumnNames()` has been added to still provide easy access to column names.

- Exception message for <sql:query> and <sql:update> has been improved. It now includes the SQL statement and provides the caught exception as the root cause.

- Warning added in <sql:transaction> about the use of commit and rollback.

- JNDI resource path to a data source must now be specified as a relative path, just as is the case in a J2EE deployment descriptor.

- New <sql:dateParam> action added to properly support setting the values of parameter markers for values of type `java.util.Date`.

- The algorithm used by the SQL actions (<sql:query>, <sql:update>, <sql:transaction>) to access a database has been modified to support configuration settings for a dataSource as well as for the JDBC DriverManager facility.

Chapters 11, 12, 13 - XML Actions

- Removed the syntax with body content for <x:set>. This was introducing a potentially confusing mechanism for entering "dynamic" XPath expressions.

- URLs specified in <x:parse> and <x:transform> may now be absolute or relative URLs.

- Clarified the fact that <x:parse> and <x:transform> do not perform any validation against DTD's or Schemas.

- XPath scopes "page", "request", "session", and "application" have been renamed "pageScope", "requestScope", "sessionScope", and "applicationScope" to be the same as the names of implicit objects in the expression language.

Appendix A - Expression Language

- Implicit objects `page`, `request`, `session`, `application`, have been renamed `pageScope`, `requestScope`, `sessionScope`, `applicationScope`.

- Implicit object `params` has been renamed `paramValues`.

- Added implicit objects `header`, `headerValues`, `cookie`, and `initParam`.

- Coercion rules have been improved.

- New operator "empty" has been added.

- "eq" and "ne" have been added as alternatives to "==" and "!="
- "&&", "||", "!" have been added as alternatives to "and", "or", and "not".

Glossary

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

action   An element in a JSP page that can act on implicit objects and other server-side objects or can expose information as JSP scoped attributes or scripting variables. Actions follow the XML syntax for elements with a start tag, a body and an end tag; if the body is empty it can also use the empty tag syntax. The tag must use a prefix.

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

scriptlet   An scripting element containing any code fragment that is valid in the scripting language used in the JSP page. The JSP specification describes what is a valid scriptlet for the case where the language page attribute is "java". Syntactically a scriptlet is delimited by the <% and %> characters.

tag   A piece of text between a left angle bracket and a right angle bracket that has a name, can have attributes, and is part of an element in a JSP page. Tag names are known to the JSP translator, either because the name is part of the JSP specification (in the case of a standard action), or because it has been introduced using a Tag Library (in the case of custom action).

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

tag library   A collection of custom actions described by a tag library descriptor and Java classes.

tag library descriptor   An XML document describing a tag library.

web application   An application built for the Internet, an intranet, or an extranet.