Expression Language Specification
Version 2.1

A component of the JavaServer™ Pages Specification
Version 2.1

Kin-Man Chung, Pierre Delisle, Mark Roth, editors
LIMITED LICENSE GRANTS

1. _License for Evaluation Purposes_. Sun hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense), under Sun's applicable intellectual property rights to view, download, use and reproduce the Specification only for the purpose of internal evaluation. This includes (i) developing applications intended to run on an implementation of the Specification, provided that such applications do not themselves implement any portion(s) of the Specification, and (ii) discussing the Specification with any third party; and (iii) excerpting brief portions of the Specification in oral or written communications which discuss the Specification provided that such excerpts do not in the aggregate constitute a significant portion of the Specification.

2. _License for the Distribution of Compliant Implementations_. Sun also grants you a perpetual, non-exclusive, non-transferable, worldwide, fully-paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights or, subject to the provisions of subsection 4 below, patent rights it may have covering the Specification to create and/or distribute an Independent Implementation of the Specification that: (a) fully implements the Specification including all its required interfaces and functionality; (b) does not modify, subset, superset or otherwise extend the Licensor Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the Licensor Name Space other than those required/authorized by the Specification or Specifications being implemented; and (c) passes the Technology Compatibility Kit (including satisfying the requirements of the applicable TCK Users Guide) for such Specification ("Compliant Implementation"). In addition, the foregoing license is expressly conditioned on your not acting outside its scope. No license is granted hereunder for any other purpose (including, for example, modifying the Specification, other than to the extent of your fair use rights, or distributing the Specification to third parties). Also, no right, title, or interest in or to any trademarks, service marks, or trade names of Sun or Sun's licensors is granted hereunder. Java, and Java-related logos, marks and names are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

3. _Pass-through Conditions_. You need not include limitations (a)-(c) from the previous paragraph or any other particular "pass through" requirements in any license You grant concerning the use of your Independent Implementation or products derived from it. However, except with respect to Independent Implementations (and products derived from them) that satisfy limitations (a)-(c) from the previous paragraph, You may neither: (a) grant or otherwise pass through to your licensees any licenses under Sun's applicable intellectual property rights; nor (b) authorize your licensees to make any claims concerning their implementation's compliance with the Specification in question.

4. _Reciprocity Concerning Patent Licenses_.

a. With respect to any patent claims covered by the license granted under subparagraph 2 above that would be infringed by all technically feasible implementations of the Specification, such license is conditioned upon your offering on fair, reasonable and non-discriminatory terms, to any party seeking it from You, a perpetual, non-exclusive, non-transferable, worldwide license under Your patent rights which are or would be infringed by all technically feasible implementations of the Specification to develop, distribute and use a Compliant Implementation.
b With respect to any patent claims owned by Sun and covered by the license granted under subparagraph 2, whether or not their infringement can be avoided in a technically feasible manner when implementing the Specification, such license shall terminate with respect to such claims if You initiate a claim against Sun that it has, in the course of performing its responsibilities as the Specification Lead, induced any other entity to infringe Your patent rights.

c Also with respect to any patent claims owned by Sun and covered by the license granted under subparagraph 2 above, where the infringement of such claims can be avoided in a technically feasible manner when implementing the Specification such license, with respect to such claims, shall terminate if You initiate a claim against Sun that its making, having made, using, offering to sell, selling or importing a Compliant Implementation infringes Your patent rights.

5. _Definitions_. For the purposes of this Agreement: "Independent Implementation” shall mean an implementation of the Specification that neither derives from any of Sun’s source code or binary code materials nor, except with an appropriate and separate license from Sun, includes any of Sun’s source code or binary code materials; “Licensor Name Space” shall mean the public class or interface declarations whose names begin with "java", "javax", "com.sun" or their equivalents in any subsequent naming convention adopted by Sun through the Java Community Process, or any recognized successors or replacements thereof; and “Technology Compatibility Kit” or “TCK” shall mean the test suite and accompanying TCK User’s Guide provided by Sun which corresponds to the Specification and that was available either (i) from Sun’s 120 days before the first release of Your Independent Implementation that allows its use for commercial purposes, or (ii) more recently than 120 days from such release but against which You elect to test Your implementation of the Specification.

This Agreement will terminate immediately without notice from Sun if you breach the Agreement or act outside the scope of the licenses granted above.

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED "AS IS". SUN MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT (INCLUDING AS A CONSEQUENCE OF ANY PRACTICE OR IMPLEMENTATION OF THE SPECIFICATION), OR THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE. This document does not represent any commitment to release or implement any portion of the Specification in any product. In addition, the Specification could include technical inaccuracies or typographical errors.

LIMITATION OF LIABILITY

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

You will indemnify, hold harmless, and defend Sun and its licensors from any claims arising or resulting from: (i) your use of the Specification; (ii) the use or distribution of your Java application, applet and/or implementation; and/or (iii) any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.
RESTRICTED RIGHTS LEGEND

U.S. Government: If this Specification is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government’s rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

REPORT

If you provide Sun with any comments or suggestions concerning the Specification (“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.

GENERAL TERMS

Any action related to this Agreement will be governed by California law and controlling U.S. federal law. The U.N. Convention for the International Sale of Goods and the choice of law rules of any jurisdiction will not apply.

The Specification is subject to U.S. export control laws and may be subject to export or import regulations in other countries. Licensee agrees to comply strictly with all such laws and regulations and acknowledges that it has the responsibility to obtain such licenses to export, re-export or import as may be required after delivery to Licensee.

This Agreement is the parties’ entire agreement relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, conditions, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification to this Agreement will be binding, unless in writing and signed by an authorized representative of each party.
Contents

1. Language Syntax and Semantics 1
   1.1 Overview 1
      1.1.1 EL in a nutshell 2
   1.2 EL Expressions 2
      1.2.1 Eval-expression 2
         1.2.1.1 Eval-expressions as value expressions 3
         1.2.1.2 Eval-expressions as method expressions 4
      1.2.2 Literal-expression 5
   1.2.3 Composite expressions 6
   1.2.4 Syntax restrictions 7
   1.3 Literals 7
   1.4 Errors, Warnings, Default Values 7
   1.5 Resolution of Model Objects and their Properties 8
1.6 Operators [ ] and . 8
1.7 Arithmetic Operators 9
  1.7.1 Binary operators - A {+, -, *} B 10
  1.7.2 Binary operator - A {/, div} B 10
  1.7.3 Binary operator - A {%, mod} B 11
  1.7.4 Unary minus operator -A 11
1.8 Relational Operators 11
  1.8.1 A {<, >, <=, >=, lt, gt, le, ge} B 12
  1.8.2 A {==, !=, eq, ne} B 12
1.9 Logical Operators 13
  1.9.1 Binary operator - A {&&, ||, and, or} B 13
1.10 Empty Operator - empty A 13
1.11 Conditional Operator - A ? B : C 14
1.12 Parentheses 14
1.13 Operator Precedence 14
1.14 Reserved Words 15
1.15 Functions 15
1.16 Variables 16
1.17 Enums 16
1.18 Type Conversion 16
  1.18.1 To Coerce a Value X to Type Y 17
  1.18.2 Coerce A to String 17
  1.18.3 Coerce A to Number type N 17
  1.18.4 Coerce A to Character 18
  1.18.5 Coerce A to Boolean 18
  1.18.6 Coerce A to an Enum Type T 19
  1.18.7 Coerce A to Any Other Type T 19
1.19 Collected Syntax 19

2. Java APIs 23
javax.el 25
   ArrayELResolver 29
   BeanELResolver 34
   BeanELResolver.BeanProperties 40
   BeanELResolver.BeanProperty 41
   CompositeELResolver 43
   ELContext 50
   ELContextEvent 54
   ELContextListener 56
   ELException 57
   ELResolver 59
   Expression 65
   ExpressionFactory 68
   FunctionMapper 72
   ListELResolver 74
   MapELResolver 79
   MethodExpression 84
   MethodInfo 87
   MethodNotFoundException 89
   PropertyNotFoundException 91
   PropertyNotWritableException 93
   ResourceBundleELResolver 95
   ValueExpression 99
   VariableMapper 103

A. Changes  105
   A.1 Changes between Porposed Final Draft and Proposed Final Draft 2  105
   A.2 Changes between Public Review and Proposed Final Draft  105
   A.3 Changes between Early Draft Release and Public Review  106
Preface

This is the Expression Language specification version 2.1, developed jointly by the JSR-245 (JSP 2.1) and JSR-252 (Faces 1.2) expert groups under the Java Community Process. See http://www.jcp.org.

Historical Note

The EL was originally inspired by both ECMAScript and the XPath expression languages. During its inception, the experts involved were very reluctant to design yet another expression language and tried to use each of these languages, but they fell short in different areas.

The JSP Standard Tag Library (JSTL) version 1.0 (based on JSP 1.2) was therefore first to introduce 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.

Given its success, the EL was subsequently moved into the JSP specification (JSP 2.0/JSTL 1.1), making it generally available within JSP pages (not just for attributes of JSTL tag libraries).

JavaServer Faces 1.0 defined a standard framework for building User Interface components, and was built on top of JSP 1.2 technology. Because JSP 1.2 technology did not have an integrated expression language and because the JSP 2.0 EL did not meet all of the needs of Faces, an EL variant was developed for Faces 1.0. The Faces expert group (EG) attempted to make the language as compatible with JSP 2.0 as possible but some differences were necessary.
It is obviously desirable to have a single, unified expression language that meets the needs of the various web-tier technologies. The Faces and JSP EGs therefore worked together on the specification of a unified expression language, defined in this document, and which takes effect for the JSP 2.1 and Faces 1.2 releases.

The JSP/JSTL/Faces expert groups also acknowledge that the Expression Language (EL) is useful beyond their own specifications. It is therefore desirable to eventually move the Expression Language into its own JSR to give it more visibility and guarantee its general applicability outside of the JSP specification.

It has not been possible to give the EL its own JSR at this time. However, as a first step, JSP 2.1 delivers 2 specification documents, one specific to the JSP technology, and one specific to the Expression Language (this document). This makes the intent clear that the Expression Language does not carry a dependency on the JSP specification and will make it easier in the future should the decision be made to move it into its own JSR.

Related Documentation

Implementors of the Expression Language and web developers may find the following documents worth consulting for additional information:

<table>
<thead>
<tr>
<th>JavaServer Pages (JSP)</th>
<th><a href="http://java.sun.com/products/jsp">http://java.sun.com/products/jsp</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaServer Faces (JSF)</td>
<td><a href="http://java.sun.com/j2ee/javaserverfaces">http://java.sun.com/j2ee/javaserverfaces</a></td>
</tr>
<tr>
<td>Java Servlet Technology</td>
<td><a href="http://java.sun.com/servlet">http://java.sun.com/servlet</a></td>
</tr>
<tr>
<td>JavaBeans</td>
<td><a href="http://java.sun.com/beans">http://java.sun.com/beans</a></td>
</tr>
</tbody>
</table>
Typographical Conventions

<table>
<thead>
<tr>
<th>Font Style</th>
<th>Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Italic</em></td>
<td>Emphasis, definition of term.</td>
</tr>
<tr>
<td><strong>Monospace</strong></td>
<td>Syntax, code examples, attribute names, Java language types, API, enumerated attribute values.</td>
</tr>
</tbody>
</table>

Acknowledgments

This specification on the unified Expression Language is the joint work of the JSR-245 (JavaServer Pages) and JSR-252 (JavaServer Faces) expert groups. We want to thank members of these two expert groups for their spirit of collaboration and excellent work on the unification of the Expression Language.

Special mention is due to Jacob Hookom and Adam Winer for their leadership role in helping solve the complex technical issues we were faced with in this unification work.

The editors also want to give special thanks to the individuals within the Java Enterprise Edition platform team at Sun Microsystems, and especially to Bill Shannon, Eduardo Pellegri-Llopart, Jim Driscoll, Karen Schaffer, Jan Luehe, Nick Rodin, Sheri Shen, Jean-Francois Arcand, Jennifer Ball, Tony Ng, Ed Burns, Jayashri Visvanathan, Roger Kitain, Ryan Lubke, Dhiru Pandey, Greg Murray, and Norbert Lindenberg.

Comments

We are interested in improving this specification and welcome your comments and suggestions. We have a java.net project with an issue tracker and a mailing list for comments and discussions about this specification:

https://jsp-spec-public.dev.java.net/
Language Syntax and Semantics

The syntax and semantics of the Expression Language (EL) are described in this chapter.

1.1 Overview

The EL is a simple language designed to meet the needs of the presentation layer in web applications. It features:

- A simple syntax restricted to the evaluation of expressions
- Variables and nested properties
- Relational, logical, arithmetic, conditional, and empty operators
- Functions implemented as static methods on Java classes
- Lenient semantics where appropriate default values and type conversions are provided to minimize exposing errors to end users

as well as

- A pluggable API for resolving variable references into Java objects and for resolving the properties applied to these Java objects
- An API for deferred evaluation of expressions that refer to either values or methods on an object
- Support for lvalue expressions (expressions a value can be assigned to)

These last three features are key additions to the JSP 2.0 EL resulting from the EL alignment work done in the JSP 2.1 and Faces 1.2 specifications.
1.1.1 EL in a nutshell

The syntax is quite simple. Model objects 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 values. Logical operators are
available.

The EL features a flexible architecture where the resolution of model objects (and
their associated properties), functions, and variables are all performed through a
pluggable API, making the EL easily adaptable to various environments.

1.2 EL Expressions

An EL expression is specified either as an eval-expression, or as a literal-expression. The
EL also supports composite expressions, where multiple EL expressions (eval-
expressions and literal-expressions) are grouped together.

An EL expression is parsed as either a value expression or a method expression. A value
expression refers to a value, whereas a method expression refers to a method on an
object. Once parsed, the expression can optionally be evaluated one or more times.

Each type of expression (eval-expression, literal-expression, and composite
expression) is described in its own section below.

1.2.1 Eval-expression

An eval-expression is formed by using the constructs ${expr} or #{expr}. Both
constructs are parsed and evaluated in exactly the same way by the EL, even though
they might carry different meanings in the technology that is using the EL.

For instance, by convention the J2EE web tier specifications use the ${expr}
construct for immediate evaluation and the #{expr} construct for deferred
evaluation. This difference in delimiters points out the semantic differences between
the two expression types in the J2EE web tier. Expressions delimited by "#"" are said
to use "deferred evaluation" because the expression is not evaluated until its value is
needed by the system. Expressions delimited by "${}" are said to use "immediate
evaluation" because the expression is compiled when the JSP page is compiled and it
is executed when the JSP page is executed. More on this in Section 1.2.4, “Syntax
restrictions”.

Other technologies may choose to use the same convention. It is up to each
technology to enforce its own restrictions on where each construct can be used.

Nested eval-expressions, such as ${item[$(i)]}, are illegal.

1.2.1.1 Eval-expressions as value expressions

When parsed as a value expression, an eval-expression can be evaluated as either an
rvalue or an lvalue. If there were an assignment operator in the EL, an rvalue is an
expression that would typically appear on the right side of the assignment operator.
An lvalue would typically appear on the left side.

For instance, all EL expressions in JSP 2.0 are evaluated by the JSP engine
immediately when the page response is rendered. They all yield rvalues.

In the following JSTL action

```xml
<c:out value="${customer.name}"/>
```

the expression ${customer.name} is evaluated by the JSP engine and the returned
value is fed to the tag handler and converted to the type associated with the
attribute (String in this case).

Faces, on the other hand, supports a full UI component model that requires
expressions to represent more than just rvalues. It needs expressions to represent
references to data structures whose value could be assigned, as well as to represent
methods that could be invoked.

For example, in the following Faces code sample:

```xml
<h:form>
    <h:inputText
        id="email"
        value="#{checkOutFormBean.email}"
        size="25" maxlength="125"
        validator="#{checkOutFormBean.validateEmail}"/>
</h:form>
```
when the form is submitted, the "apply request values" phase of Faces evaluates the EL expression #{checkOutFormBean.email} as a reference to a data structure whose value is set with the input parameter it is associated with in the form. The result of the expression therefore represents a reference to a data structure, or an lvalue, the left hand side of an assignment operation.

When that same expression is evaluated during the rendering phase, it yields the specific value associated with the object (rvalue), just as would be the case with JSP.

The valid syntax for an lvalue is a subset of the valid syntax for an rvalue. In particular, an lvalue can only consist of either a single variable (e.g. ${name}) or a property resolution on some object, via the . or [] operator (e.g. ${employee.name}).

When parsing a value expression, an expected type is provided. In the case of an rvalue, the expected type is what the result of the expression evaluation is coerced to. In the case of lvalues, the expected type is ignored and the provided value is coerced to the actual type of the property the expression points to, before that property is set. The EL type conversion rules are defined in Section 1.18, “Type Conversion”. A few sample eval-expressions are shown in FIGURE 1-1.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Expected Type</th>
<th>Result</th>
</tr>
</thead>
</table>
| ${customer.name}    | String        | Guy Lafleur
Expression evaluates to a String. No conversion necessary.         |
| ${book}             | String        | Wonders of the World
Expression evaluates to a Book object (e.g. com.example.Book). Conversion rules result in the evaluation of book.toString(), which could for example yield the book title. |

FIGURE 1-1 Sample eval-expressions

1.2.1.2 Eval-expressions as method expressions

In some cases, it is desirable for an EL expression to refer to a method instead of a model object.

For instance, in JSF, a component tag also has a set of attributes for referencing methods that can perform certain functions for the component associated with the tag. To support these types of expressions, the EL defines method expressions (EL class MethodExpression).
In the above example, the validator attribute uses an expression that is associated
with type MethodExpression. Just as with ValueExpressions, the evaluation of
the expression (calling the method) is deferred and can be processed by the
underlying technology at the appropriate moment within its lifecycle.

A method expression shares the same syntax as an lvalue. That is, it can only consist
of either a single variable (e.g. ${name}) or a property resolution on some object, via
the . or [] operator (e.g. ${employee.name}). Information about the expected
return type and parameter types is provided at the time the method is provided.

A method expression is evaluated by invoking its referenced method or by
retrieving information about the referenced method. Upon evaluation, the EL API
verifies that the method conforms to the expected signature provided at parse time.
There is therefore no coercion performed by method expressions.

1.2.2 Literal-expression

A literal-expression does not use the $expr or #{expr} constructs, and simply
evaluates to the text of the expression, of type String. Upon evaluation, an
expected type of something other than String can be provided. Sample literal-
expressions are shown in FIGURE 1-2.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Expected Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aloha!</td>
<td>String</td>
<td>Aloha!</td>
</tr>
<tr>
<td>true</td>
<td>Boolean</td>
<td>Boolean.TRUE</td>
</tr>
</tbody>
</table>

FIGURE 1-2 Sample literal-expressions

To generate literal values that include the character sequence "${" or "#{", the
developer can choose to use a composite expression as shown here:

`${exprA}

#{exprB}
The resulting values would then be the strings ${exprA} and #{exprB}.

Alternatively, the escape characters \$ and \# can be used to escape what would
otherwise be treated as an eval-expression. Given the literal-expressions:

\${exprA}

\#{exprB}
The resulting values would again be the strings ${exprA} and #{exprB}.
A literal-expression can be used anywhere a value expression can be used. A literal-expression can also be used as a method expression that returns a non-void return value. The standard EL coercion rules (see Section 1.18, “Type Conversion”) then apply if the return type of the method expression is not java.lang.String.

### 1.2.3 Composite expressions

The EL also supports composite expressions, where multiple EL expressions are grouped together. With composite expressions, eval-expressions are evaluated from left to right, coerced to strings (according to the EL type conversion rules), and concatenated with any intervening literal-expressions.

For example, the composite expression "${firstName} ${lastName}” is composed of three EL expressions: eval-expression "${firstName}”, literal-expression “ “, and eval-expression "${lastName}". Once evaluated, the resulting string is then coerced to the expected type, according to the EL type conversion rules. A sample composite expression is shown in FIGURE 1-3.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Expected Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Welcome</td>
<td>String</td>
<td>Welcome Guy Lafleur to our site</td>
</tr>
<tr>
<td>$(customer.name) to our site</td>
<td>$(customer.name) evaluates to a string which is then concatenated with the literal-expressions. No conversion necessary.</td>
<td></td>
</tr>
</tbody>
</table>

**FIGURE 1-3** Sample composite expression

It is illegal to mix ${} and #{} constructs in a composite expression. This restriction is imposed to avoid ambiguities should a user think that using ${expr} or #{expr} dictates how an expression is evaluated. For instance, as was mentioned previously, the convention in the J2EE web tier specifications is for ${} to mean immediate evaluation and for #{} to mean deferred evaluation. This means that in EL expressions in the J2EE web tier, a developer cannot force immediate evaluation of some parts of a composite expression and deferred evaluation of other parts. This restriction may be lifted in future versions to allow for more advanced EL usage patterns.

A composite expression can be used anywhere an EL expression can be used except for when parsing a method expression. Only a single eval-expression can be used to parse a method expression.
1.2.4 Syntax restrictions

While $()$ and #() eval-expressions are parsed and evaluated in exactly the same way by the EL, the underlying technology is free to impose restrictions on which syntax can be used according to where the expression appears.

For instance, in JSP 2.1, #() expressions are only allowed for tag attributes that accept deferred expressions. #{expr} will generate an error if used anywhere else.

1.3 Literals

There are literals for boolean, integer, floating point, string, and null in an eval-expression.
- Boolean - true and false
- Integer - As defined by the IntegerLiteral construct in Section 1.19
- Floating point - As defined by the FloatingPointLiteral construct in Section 1.19
- 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

1.4 Errors, Warnings, Default Values

The Expression Language has been designed with the presentation layer of web applications in mind. 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 environment where the EL is used).
1.5 Resolution of Model Objects and their Properties

A core concept in the EL is the evaluation of a model object name into an object, and the resolution of properties applied to objects in an expression (operators . and []).

The EL API provides a generalized mechanism, an ELResolver, implemented by the underlying technology and which defines the rules that govern the resolution of model object names and their associated properties.

1.6 Operators [ ] and .

The EL follows ECMAScript in unifying the treatment of the . and [] operators.

expr-a.identifier-b is equivalent to expr-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:
  - If expr-a[expr-b] is the last property being resolved:
    - If the expression is a value expression and ValueExpression.getValue(context) was called to initiate this expression evaluation, return null.
    - Otherwise, throw PropertyNotFoundException. [trying to de-reference null for an lvalue]
  - Otherwise, return null.
- Evaluate expr-b into value-b.
- If value-b is null:
  - If expr-a[expr-b] is the last property being resolved:
    - If the expression is a value expression and ValueExpression.getValue(context) was called to initiate this expression evaluation, return null.
    - Otherwise, throw PropertyNotFoundException. [trying to de-reference null for an lvalue]
- Otherwise, return null.
- If the expression is a value expression:
  - If expr-a[expr-b] is the last property being resolved:
    - If ValueExpression.getValue(context) was called to initiate this expression evaluation, invoke elResolver.getValue(context, value-a, value-b).
    - If ValueExpression.getType(context) was called, invoke elResolver.getType(context, value-a, value-b).
    - If ValueExpression.isReadOnly(context) was called, invoke elResolver.isReadOnly(context, value-a, value-b).
    - If ValueExpression.setValue(context, val) was called, invoke elResolver.setValue(context, value-a, value-b, val).
  - Otherwise:
    - Invoke elResolver.getValue(value-a, value-b).
- Otherwise, the expression is a method expression:
  - If expr-a[expr-b] is the last property being resolved:
    - Coerce value-b to String.
    - Find the method on object value-a with name value-b and with the set of expected parameter types provided at parse time. If the method does not exist, or the return type does not match the expected return type provided at parse time, throw MethodNotFoundException.
    - If MethodExpression.invoke(context, params) was called, invoke the found method with the parameters passed to the invoke method.
    - If MethodExpression.getMethodInfo(context) was called, construct and return a new MethodInfo object.
  - Otherwise:
    - Invoke elResolver.getValue(value-a, value-b).

### 1.7 Arithmetic Operators

Arithmetic is provided to act on integer (BigInteger and Long) and floating point (BigDecimal and Double) values. There are 5 operators:

- Addition: +
- Subtraction: -
- 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.

1.7.1 Binary operators - A \{+, -, *\} B

- If A and B are null, return (Long) 0
- If A or B is a BigDecimal, coerce both to BigDecimal and then:
  - If operator is +, return A.add(B)
  - If operator is -, return A.subtract(B)
  - If operator is *, return A.multiply(B)
- If A or B is a Float, Double, or String containing ., e, or E:
  - If A or B is BigInteger, coerce both A and B to BigDecimal and apply operator.
  - Otherwise, coerce both A and B to Double and apply operator.
- If A or B is BigInteger, coerce both to BigInteger and then:
  - If operator is +, return A.add(B)
  - If operator is -, return A.subtract(B)
  - If operator is *, return A.multiply(B)
- Otherwise coerce both A and B to Long and apply operator.
- If operator results in exception, error.

1.7.2 Binary operator - A \{/, div\} B

- If A and B are null, return (Long) 0
- If A or B is a BigDecimal or a BigInteger, coerce both to BigDecimal and return A.divide(B, BigDecimal.ROUND_HALF_UP)
- Otherwise, coerce both A and B to Double and apply operator.
- If operator results in exception, error.
1.7.3 Binary operator - A \( \% , \mod \) B

- If A and B are null, return (Long) 0
- If A or B is a BigDecimal, Float, Double, or String containing ., e, or E, coerce both A and B to Double and apply operator
- If A or B is a BigInteger, coerce both to BigInteger and return A.modulo(B).
- Otherwise coerce both A and B to Long and apply operator
- If operator results in exception, error

1.7.4 Unary minus operator - \(-A\)

- If A is null, return (Long) 0
- If A is a BigDecimal or BigInteger, return A.negate().
- 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

1.8 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.

1.8.1 A {<,>,<,,>=,lt,gt,le,ge} B

- If A==B, if operator is <, le, ge, return true.
- If A is null or B is null, return false
- If A or B is BigDecimal, coerce both A and B to BigDecimal and use the return value of A.compareTo(B).
- If A or B is Float or Double, coerce both A and B to Double, apply operator
- If A or B is BigInteger, coerce both A and B to BigInteger and use the return value of A.compareTo(B).
- 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, then:
  - If A.compareTo(B) throws exception, error.
  - Otherwise use result of A.compareTo(B)
- If B is Comparable, then:
  - If B.compareTo(A) throws exception, error.
  - Otherwise use result of B.compareTo(A)
- Otherwise, error

1.8.2 A {==, !=, eq, ne} B

- If A==B, apply operator
- If A is null or B is null, return false for == or eq, true for != or ne.
- If A or B is BigDecimal, coerce both A and B to BigDecimal and then:
  - If operator is == or eq, return A.equals(B)
  - If operator is != or ne, return !A.equals(B)
- If A or B is Float or Double, coerce both A and B to Double, apply operator
- If A or B is BigInteger, coerce both A and B to BigInteger and then:
  - If operator is == or eq, return A.equals(B)
  - If operator is != or ne, return !A.equals(B)
1.9 Logical Operators

The logical operators are:

- `&&` and `and`
- `||` and `or`
- `!` and `not`

The evaluation of logical operators is described in the following sections.

1.9.1 Binary operator - `A {&&, ||, and, or} B`

- Coerce both `A` and `B` to `Boolean`, apply operator

The operator stops as soon as the expression can be determined, i.e., `A and B and C and D` - if `B` is false, then only `A and B` is evaluated. Unary not operator - `{!,not} A`
- Coerce `A` to `Boolean`, apply operator

1.10 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 Collection, return true
Otherwise return false

1.11 Conditional Operator - A ? B : C
Evaluate B or C, depending on the result of the evaluation of A.
Coerce A to Boolean:
■ If A is true, evaluate and return B
■ If A is false, evaluate and return C

1.12 Parentheses
Parentheses can be used to change precedence, as in: $(a*(b+c))$

1.13 Operator Precedence
Highest to lowest, left-to-right.
■ []
■ ()
■ -(unary) not ! empty
■ * / div % mod
■ + - (binary)
■ < > <= >= lt gt le ge
■ == != eq ne
■ && and
■ || or
■ ? :
Qualified functions with a namespace prefix have precedence over the operators. Thus the expression ${c?b:f()} is illegal because b:f() is being parsed as a qualified function instead of part of a conditional expression. As usual, () can be used to make the precedence explicit, e.g. ${c?b:(f())}

1.14 Reserved Words

The following words are reserved for the language and must not be used as identifiers.

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 must avoid using these words.

1.15 Functions

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

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

$[ns:]f([a_1],[a_2],\ldots,[a_n])$

Where ns is the namespace prefix, f is the name of the function, and a is an argument.

EL functions are mapped, resolved and bound at parse time. It is the responsibility of the FunctionMapper class to provide the mapping of namespace-qualified functions to static methods of specific classes when expressions are created. If no FunctionMapper is provided (by passing in null), functions are disabled.
1.16 Variables

Just like FunctionMapper provides a flexible mechanism to add functions to the EL, VariableMapper provides a flexible mechanism to support the notion of EL variables. An EL variable does not directly refer to a model object that can then be resolved by an ELResolver. Instead, an EL variable refers to an EL expression. The evaluation of that EL expression yields the value associated with the EL variable.

EL variables are mapped, resolved and bound at parse time. It is the responsibility of the VariableMapper class to provide the mapping of EL variables to ValueExpressions when expressions are created. If no VariableMapper is provided (by passing in null), variable mapping is disabled.

See the javax.el package description for more details.

1.17 Enums

The Unified EL supports Java SE 5 enumerated types. Coercion rules for dealing with enumerated types are included in the following section. Also, when referring to values that are instances of an enumerated type from within an EL expression, use the literal string value to cause coercion to happen via the below rules. For example, let’s say we have an enum called Suit that has members Heart, Diamond, Club, and Spade. Furthermore, let’s say we have a reference in the EL, mySuit, that is a Spade. If you want to test for equality with the Spade enum, you would say ${mySuit == 'Spade'}. The type of the mySuit will trigger the invocation of Enum.valueOf(Suit.class, 'Spade').

1.18 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.
1.18.1 To Coerce a Value X to Type Y

- If X is of a primitive type, Let X’ be the equivalent “boxed form” of X. Otherwise, Let X’ be the same as X.
- If Y is of a primitive type, Let Y’ be the equivalent “boxed form” of Y. Otherwise, Let Y’ be the same as Y.
- Apply the rules in Sections 1.18.2-1.18.7 for coercing X’ to Y’.
- If Y is a primitive type, then the result is found by "unboxing" the result of the coercion. If the result of the coercion is null, then error.
- If Y is not a primitive type, then the result is the result of the coercion.

For example, if coercing an int to a String, "box" the int into an Integer and apply the rule for coercing an Integer to a String. Or if coercing a String to a double, apply the rule for coercing a String to a Double, then "unbox" the resulting Double, making sure the resulting Double isn't actually null.

1.18.2 Coerce A to String

- If A is String: return A
- Otherwise, if A is null: return ""
- Otherwise, if A is Enum, return A.name()
- Otherwise, if A.toString() throws an exception, error
- Otherwise, return A.toString()

1.18.3 Coerce A to Number type N

- If A is null or "", return 0.
- If A is Character, convert A to new Short((short)a.charValue()), and apply the following rules.
- If A is Boolean, then error.
- If A is Number type N, return A
- If A is Number, coerce quietly to type N using the following algorithm:
  - If N is BigInteger
    - If A is a BigDecimal, return A.toBigInteger()
    - Otherwise, return BigInteger.valueOf(A.longValue())
  - If N is BigDecimal,
    - If A is a BigInteger, return new BigDecimal(A)
    - Otherwise, return new BigDecimal(A.doubleValue())
  - If N is Byte, return new Byte(A.byteValue())
If \( N \) is Short, return new Short(A.shortValue())
If \( N \) is Integer, return new Integer(A.intValue())
If \( N \) is Long, return new Long(A.longValue())
If \( N \) is Float, return new Float(A.floatValue())
If \( N \) is Double, return new Double(A.doubleValue())
Otherwise, error.
If \( A \) is String, then:
  If \( N \) is BigDecimal then:
    If new BigDecimal(A) throws an exception then error.
    Otherwise, return new BigDecimal(A).
  If \( N \) is BigInteger then:
    If new BigInteger(A) throws an exception then error.
    Otherwise, return new BigInteger(A).
    If \( N \).valueOf(A) throws an exception, then error.
    Otherwise, return \( N \).valueOf(A).
  Otherwise, error.

1.18.4 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, coerce quietly to type Short, then return a Character whose numeric value is equivalent to that of a Short.
- If \( A \) is String, return \( A \).charAt(0)
- Otherwise, error

1.18.5 Coerce \( A \) to Boolean
- If \( A \) is null or "", return false
- Otherwise, if \( A \) is a Boolean, return \( A \)
- Otherwise, if \( A \) is a String, and Boolean.valueOf(A) does not throw an exception, return it
1.18.6 Otherwise, error Coerce A to an Enum Type T

- If A is null, return null
- If A is assignable to T, coerce quietly
- If A is "", return null.

1.18.7 If A is a String call Enum.valueOf(T.getClass(), A) and return the result. 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 String, and T has no PropertyEditor:
  - If A is ", return null
  - Otherwise error
- If A is a String and T's PropertyEditor throws an exception:
  - If A is ", return null
  - Otherwise, error
- Otherwise, apply T's PropertyEditor
- Otherwise, error

1.19 Collected Syntax

The valid syntax for an expression depends on its type.

For value expressions, the parser first attempts to parse the expression using the LValue production. If parsing fails, the ValueExpression will be read-only and parsing is attempted again using the RValue production. For method expressions, the parser must use only the MethodExpression production.

These productions take into consideration literal-expressions and composite expressions wherever they are accepted.

\[
\text{LValue} ::= \quad '{' LValueInner '}'
          | '#(' LValueInner ')
\]
LValueInner ::= Identifier
| NonLiteralValuePrefix (ValueSuffix)*

RValue ::= (RValueComponent1)+
| (RValueComponent2)+

RValueComponent1 ::= '${' Expression '}'
| LiteralExpression

RValueComponent2 ::= '#{' Expression '}'
| LiteralExpression

MethodExpression ::= LValue

LiteralExpression ::= (LiteralComponent)* ([#])?
  i.e., a string of any characters that
  doesn’t include ${ or #{ unless escaped by
  \${ or \#{.

LiteralComponent ::= (^[#\])*(^[#])?
| (^[#])*(^[#][^[}])
| ([#])*

Expression ::= Expression1 ExpressionRest?

ExpressionRest ::= '?' Expression ':' Expression

Expression1 ::= Expression BinaryOp Expression
| UnaryExpression

BinaryOp ::= 'and'
| '&&'
| 'or'
| '||'
| '+'
| '-'
| '*'
| '/'
| 'div'
| '%'
| 'mod'
| '>'
| 'gt'
| '<'
| 'lt'
| '>='
| 'ge'
| '<='
| 'le'
| '=='
| 'eq'
| '!='
| 'ne'

UnaryExpression ::= UnaryOp UnaryExpression
| Value
UnaryOp ::= '−' | '!' | 'not' | 'empty'

Value ::= ValuePrefix (ValueSuffix)*

ValuePrefix ::= Literal | NonLiteralValuePrefix

NonLiteralValuePrefix ::= '(' Expression ')' | Identifier | FunctionInvocation

ValueSuffix ::= '.' Identifier | '[(' Expression ')]'

Identifier ::= Java language identifier

FunctionInvocation ::= (Identifier ':'? Identifier '(' (Expression (',' Expression )*?)? ')'?

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 ::= ['0'-'9']+

FloatingPointLiteral ::= ([0-'9']+) ('.' ([0-'9'])* Exponent? | 'e' ('E')? Exponent?)

Exponent ::= ['e','E'] (['+','-']? ([0-'9']*)

NullLiteral ::= 'null'

Notes

• * = 0 or more, + = 1 or more, ? = 0 or 1.
• An identifier is constrained to be a Java identifier - e.g., no -, no /, etc.
• A String only recognizes a limited set of escape sequences, and \ may not appear unescaped.
■ The relational operator for equality is == (double equals).

■ The value of an IntegerLiteral ranges from Long.MIN_VALUE to Long.MAX_VALUE

■ The value of a FloatingPointLiteral ranges from Double.MIN_VALUE to Double.MAX_VALUE

■ It is illegal to nest $( or #{ inside an outer $( or #{.
CHAPTER 2

Java APIs

This chapter describes the Java APIs exposed by the EL specification. The content of this chapter is generated automatically from Javadoc annotations embedded into the actual Java classes and interfaces of the implementation. This ensures that both the specification and implementation are synchronized.
2.0 Package
javax.el

2.0.1 Description
Provides the API for the Unified Expression Language shared by the JSP 2.1 and JSF 1.2 technologies.

The Expression Language (EL) is a simple language designed to satisfy the specific needs of web application developers. It is currently defined in its own specification document within the JavaServer Pages (tm) (JSP) 2.1 specification, but does not have any dependencies on any portion of the JSP 2.1 specification. It is intended for general use outside of the JSP and JSF specifications as well.

This package contains the classes and interfaces that describe and define the programmatic access to the Expression Language engine. The API is logically partitioned as follows:

- EL Context
- Expression Objects
- Creation of Expressions
- Resolution of Model Objects and their Properties
- EL Functions
- EL Variables

2.0.2 EL Context
An important goal of the EL is to ensure it can be used in a variety of environments. It must therefore provide enough flexibility to adapt to the specific requirements of the environment where it is being used.

Class ELContext is what links the EL with the specific environment where it is being used. It provides the mechanism through which all relevant context for creating or evaluating an expression is specified.

Creation of ELContext objects is controlled through the underlying technology. For example, in JSP, the JspContext.getELContext() factory method is used.

Some technologies provide the ability to add an ELContextListener so that applications and frameworks can ensure their own context objects are attached to any newly created ELContext.

2.0.3 Expression Objects
At the core of the Expression Language is the notion of an expression that gets parsed according to the grammar defined by the Expression Language.

There are two types of expressions defined by the EL: value expressions and method expressions. A ValueExpression such as "${customer.name}" can be used either as an rvalue (return the value associated with property name of the model object customer) or as an lvalue (set the value of the property name of the model object customer).

A MethodExpression such as "${handler.process}" makes it possible to invoke a method (process) on a specific model object (handler).

All expression classes extend the base class Expression, making them serializable and forcing them to implement equals() and hashCode(). Moreover, each method on these expression classes that actually evaluates an expression receives a parameter of class ELContext, which provides the context required to evaluate the expression.
2.0.4 Creation of Expressions
An expression is created through the `ExpressionFactory` class. The factory provides two creation methods; one for each type of expression supported by the EL.

To create an expression, one must provide an `ELContext`, a string representing the expression, and the expected type (ValueExpression) or signature (MethodExpression). The ELContext provides the context necessary to parse an expression. Specifically, if the expression uses an EL function (for example `${fn:toUpperCase(customer.name)}`) or an EL variable, then `FunctionMapper` and `VariableMapper` objects must be available within the ELContext so that EL functions and EL variables are properly mapped.

2.0.5 Resolution of Model Objects and their Properties
Through the `ELResolver` base class, the EL features a pluggable mechanism to resolve model object references as well as properties of these objects.

The EL API provides implementations of `ELResolver` supporting property resolution for common data types which include arrays (`ArrayELResolver`), JavaBeans (`BeanELResolver`), Lists (`ListELResolver`), Maps (`MapELResolver`), and Resource Bundles (`ResourceBundleELResolver`).

Tools can easily obtain more information about resolvable model objects and their resolvable properties by calling method `getFeatureDescriptors` on the `ELResolver`. This method exposes objects of type `java.beans.FeatureDescriptor`, providing all information of interest on top-level model objects as well as their properties.

2.0.6 EL Functions
If an EL expression uses a function (for example `${fn:toUpperCase(customer.name)}`), then a `FunctionMapper` object must also be specified within the ELContext. The `FunctionMapper` is responsible to map `${prefix:name()}` style functions to static methods that can execute the specified functions.

2.0.7 EL Variables
Just like `FunctionMapper` provides a flexible mechanism to add functions to the EL, `VariableMapper` provides a flexible mechanism to support the notion of EL variables.

An EL variable does not directly refer to a model object that can then be resolved by an ELResolver. Instead, it refers to an EL expression. The evaluation of that EL expression gives the EL variable its value.

For example, in the following code snippet

```xml
<h:inputText value="#{handler.customer.name}"/>
```

`handler` refers to a model object that can be resolved by an EL Resolver.

However, in this other example:

```xml
<c:forEach var="item" items="#{model.list}">
    <h:inputText value="#{item.name}"/>
</c:forEach>
```

`item` is an EL variable because it does not refer directly to a model object. Instead, it refers to another EL expression, namely a specific item in the collection referred to by the EL expression `#{model.list}`.

Assuming that there are three elements in `${model.list}`, this means that for each invocation of `<h:inputText>`, the following information about `item` must be preserved in the `VariableMapper`:

- first invocation: `item` maps to first element in `${model.list}`
- second invocation: `item` maps to second element in `${model.list}`

```xml
valuation
```
third invocation: item maps to third element in ${model.list}

VariableMapper provides the mechanisms required to allow the mapping of an EL variable to the EL expression from which it gets its value.

### Class Summary

<table>
<thead>
<tr>
<th>Interfaces</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELContextListener</td>
<td>The listener interface for receiving notification when an ELContext is created.</td>
</tr>
<tr>
<td>ELResolver</td>
<td>Defines property resolution behavior on arrays.</td>
</tr>
<tr>
<td>BeanELResolver</td>
<td>Defines property resolution behavior on objects using the JavaBeans component architecture.</td>
</tr>
<tr>
<td>CompositeELResolver</td>
<td>Maintains an ordered composite list of child ELResolvers.</td>
</tr>
<tr>
<td>ELContext</td>
<td>Context information for expression evaluation.</td>
</tr>
<tr>
<td>ELContextEvent</td>
<td>An event which indicates that an ELContext has been created.</td>
</tr>
<tr>
<td>ELResolver</td>
<td>Enables customization of variable and property resolution behavior for EL expression evaluation.</td>
</tr>
<tr>
<td>Expression</td>
<td>Base class for the expression subclasses ValueExpression and MethodExpression, implementing characteristics common to both.</td>
</tr>
<tr>
<td>ExpressionFactory</td>
<td>Parses a String into a ValueExpression or MethodExpression instance for later evaluation.</td>
</tr>
<tr>
<td>FunctionMapper</td>
<td>The interface to a map between EL function names and methods.</td>
</tr>
<tr>
<td>ListELResolver</td>
<td>Defines property resolution behavior on instances of java.util.List.</td>
</tr>
<tr>
<td>MapELResolver</td>
<td>Defines property resolution behavior on instances of java.util.Map.</td>
</tr>
<tr>
<td>MethodExpression</td>
<td>An Expression that refers to a method on an object.</td>
</tr>
<tr>
<td>MethodInfo</td>
<td>Holds information about a method that a MethodExpression evaluated to.</td>
</tr>
<tr>
<td>ResourceBundleELResolver</td>
<td>Defines property resolution behavior on instances of java.util.ResourceBundle.</td>
</tr>
<tr>
<td>ValueExpression</td>
<td>An Expression that can get or set a value.</td>
</tr>
<tr>
<td>VariableMapper</td>
<td>The interface to a map between EL variables and the EL expressions they are associated with.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exceptions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELException</td>
<td>Represents any of the exception conditions that can arise during expression evaluation.</td>
</tr>
</tbody>
</table>
### MethodNotFoundException
Thrown when a method could not be found while evaluating a `MethodExpression`.

### PropertyNotFoundException
Thrown when a property could not be found while evaluating a `ValueExpression` or `MethodExpression`.

### PropertyNotWritableException
Thrown when a property could not be written to while setting the value on a `ValueExpression`.

<table>
<thead>
<tr>
<th>Class Summary</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MethodNotFoundException</td>
<td>Thrown when a method could not be found while evaluating a <code>MethodExpression</code>.</td>
</tr>
<tr>
<td>PropertyNotFoundException</td>
<td>Thrown when a property could not be found while evaluating a <code>ValueExpression</code> or <code>MethodExpression</code>.</td>
</tr>
<tr>
<td>PropertyNotWritableException</td>
<td>Thrown when a property could not be written to while setting the value on a <code>ValueExpression</code>.</td>
</tr>
</tbody>
</table>
2.1 javax.el

ArrayELResolver

2.1.1 Declaration

public class ArrayELResolver extends ELResolver

java.lang.Object
    |---javax.el.ELResolver
    |   |---javax.el.ArrayELResolver

2.1.2 Description

Defines property resolution behavior on arrays.

This resolver handles base objects that are Java language arrays. It accepts any object as a property and coerces that object into an integer index into the array. The resulting value is the value in the array at that index.

This resolver can be constructed in read-only mode, which means that isReadOnly will always return true and setValue(ELContext, Object, Object, Object) will always throw PropertyNotWritableException.

ELResolvers are combined together using CompositeELResolver, to define rich semantics for evaluating an expression. See the javadocs for ELResolver for details.

Since: JSP 2.1

See Also: CompositeELResolver, ELResolver

Member Summary

<table>
<thead>
<tr>
<th>Constructors</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayELResolver()</td>
<td>getCommonPropertyType(ELContext context, java.lang.Object base)</td>
</tr>
<tr>
<td>ArrayELResolver(boolean isReadOnly)</td>
<td>getFeatureDescriptors(ELContext context, java.lang.Object base)</td>
</tr>
<tr>
<td></td>
<td>getType(ELContext context, java.lang.Object base, java.lang.Object property)</td>
</tr>
<tr>
<td></td>
<td>getValue(ELContext context, java.lang.Object base, java.lang.Object property)</td>
</tr>
<tr>
<td></td>
<td>isReadOnly(ELContext context, java.lang.Object base, java.lang.Object property)</td>
</tr>
</tbody>
</table>
2.1.3 ArrayELResolver()

public ArrayELResolver()

Creates a new read/write ArrayELResolver.

2.1.4 ArrayELResolver(boolean)

public ArrayELResolver(boolean isReadOnly)

Creates a new ArrayELResolver whose read-only status is determined by the given parameter.

Parameters:
- isReadOnly - true if this resolver cannot modify arrays; false otherwise.

2.1.5 getCommonPropertyType(ELContext, Object)

public java.lang.Class<?> getCommonPropertyType(javax.el.ELContext context, java.lang.Object base)

If the base object is a Java language array, returns the most general type that this resolver accepts for the property argument. Otherwise, returns null.

Assuming the base is an array, this method will always return Integer.class. This is because arrays accept integers for their index.

Overrides: getCommonPropertyType in class ELResolver

Parameters:
- context - The context of this evaluation.
- base - The array to analyze. Only bases that are a Java language array are handled by this resolver.

Returns: null if base is not a Java language array; otherwise Integer.class.

2.1.6 getFeatureDescriptors(ELContext, Object)

public java.util.Iterator<FeatureDescriptor> getFeatureDescriptors(javax.el.ELContext context, java.lang.Object base)

Always returns null, since there is no reason to iterate through set set of all integers.
The `getCommonPropertyType(ELContext, Object)` method returns sufficient information about what properties this resolver accepts.

**Overrides:** `getFeatureDescriptors` in class `ELResolver`

**Parameters:**
- `context` - The context of this evaluation.
- `base` - The array to analyze. Only bases that are a Java language array are handled by this resolver.

**Returns:** `null`.

### 2.1.7 `getType(ELContext, Object, Object)`

```java
public java.lang.Class<?>
getType(javax.el.ELContext context, java.lang.Object base, java.lang.Object property)
```

If the base object is an array, returns the most general acceptable type for a value in this array.

If the base is an array, the `propertyResolved` property of the `ELContext` object must be set to `true` by this resolver, before returning. If this property is not `true` after this method is called, the caller should ignore the return value.

Assuming the base is an array, this method will always return `base.getClass().getComponentType()`, which is the most general type of component that can be stored at any given index in the array.

**Overrides:** `getType` in class `ELResolver`

**Parameters:**
- `context` - The context of this evaluation.
- `base` - The array to analyze. Only bases that are Java language arrays are handled by this resolver.
- `property` - The index of the element in the array to return the acceptable type for. Will be coerced into an integer, but otherwise ignored by this resolver.

**Returns:** If the `propertyResolved` property of `ELContext` was set to `true`, then the most general acceptable type; otherwise undefined.

**Throws:**
- `PropertyNotFoundException` - if the given index is out of bounds for this array.
- `java.lang.NullPointerException` - if `context` is `null`
- `ELException` - if an exception was thrown while performing the property or variable resolution. The thrown exception must be included as the cause property of this exception, if available.

### 2.1.8 `getValue(ELContext, Object, Object)`

```java
public java.lang.Object
getValue(javax.el.ELContext context, java.lang.Object base, java.lang.Object property)
```

If the base object is a Java language array, returns the value at the given index. The index is specified by the `property` argument, and coerced into an integer. If the coercion could not be performed, an `IllegalArgumentException` is thrown. If the index is out of bounds, `null` is returned.

If the base is a Java language array, the `propertyResolved` property of the `ELContext` object must be set to `true` by this resolver, before returning. If this property is not `true` after this method is called, the caller should ignore the return value.

**Overrides:** `getValue` in class `ELResolver`
2.1.9  isReadOnly(ELContext, Object, Object)

public boolean isReadOnly(javax.el.ELContext context, java.lang.Object base, java.lang.Object property)

If the base object is a Java language array, returns whether a call to setValue(ELContext, Object, Object) will always fail.

If the base is a Java language array, the propertyResolved property of the ELContext object must be set to true by this resolver, before returning. If this property is not true after this method is called, the caller should ignore the return value.

If this resolver was constructed in read-only mode, this method will always return true. Otherwise, it returns false.

Overrides:  isReadOnly in class ELResolver

Parameters:

context - The context of this evaluation.
base - The array to analyze. Only bases that are a Java language array are handled by this resolver.
property - The index of the element in the array to return the acceptable type for. Will be coerced into an integer, but otherwise ignored by this resolver.

Returns: If the propertyResolved property of ELContext was set to true, then true if calling the setValue method will always fail or false if it is possible that such a call may succeed; otherwise undefined.

Throws:

PropertyNotFoundException - if the given index is out of bounds for this array.
java.lang.NullPointerException - if context is null
ELException - if an exception was thrown while performing the property or variable resolution. The thrown exception must be included as the cause property of this exception, if available.
If the base object is a Java language array, attempts to set the value at the given index with the given value. The index is specified by the **property** argument, and coerced into an integer. If the coercion could not be performed, an **IllegalArgumentException** is thrown. If the index is out of bounds, a **PropertyNotFoundException** is thrown.

If the base is a Java language array, the **propertyResolved** property of the **ELContext** object must be set to true by this resolver, before returning. If this property is not true after this method is called, the caller can safely assume no value was set.

If this resolver was constructed in read-only mode, this method will always throw **PropertyNotWritableException**.

**Overrides:** `setValue(int)` in class `ELResolver

**Parameters:**
- `context` - The context of this evaluation.
- `base` - The array to be modified. Only bases that are Java language arrays are handled by this resolver.
- `property` - The index of the value to be set. Will be coerced into an integer.
- `val` - The value to be set at the given index.

**Throws:**
- `java.lang.ClassCastException` - if the class of the specified element prevents it from being added to this array.
- `java.lang.NullPointerException` - if `context` is null.
- `java.lang.IllegalArgumentException` - if the property could not be coerced into an integer, or if some aspect of the specified element prevents it from being added to this array.
- `PropertyNotWritableException` - if this resolver was constructed in read-only mode.
- `ELException` - if an exception was thrown while performing the property or variable resolution. The thrown exception must be included as the cause property of this exception, if available.
2.2 javax.el BeanELResolver

2.2.1 Declaration
public class BeanELResolver extends ELResolver

java.lang.Object
   |---javax.el.ELResolver
      |---javax.el.BeanELResolver

2.2.2 Description
Defines property resolution behavior on objects using the JavaBeans component architecture.
This resolver handles base objects of any type, as long as the base is not null. It accepts any object as a property, and coerces it to a string. That string is then used to find a JavaBeans compliant property on the base object. The value is accessed using JavaBeans getters and setters.
This resolver can be constructed in read-only mode, which means that isReadOnly will always return true and setValue(ELContext, Object, Object, Object) will always throw PropertyNotWritableException.
ELResolvers are combined together using CompositeELResolver's, to define rich semantics for evaluating an expression. See the javadocs for ELResolver for details.
Because this resolver handles base objects of any type, it should be placed near the end of a composite resolver. Otherwise, it will claim to have resolved a property before any resolvers that come after it get a chance to test if they can do so as well.

Since: JSP 2.1

See Also: CompositeELResolver, ELResolver

Member Summary

<table>
<thead>
<tr>
<th>Nested Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>protected static class BeanELResolver.BeanProperties</td>
</tr>
<tr>
<td>protected static class BeanELResolver.BeanProperty</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeanELResolver()</td>
</tr>
<tr>
<td>BeanELResolver(boolean isReadOnly)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.Class&lt;?&gt; getCommonPropertyType(ELContext context, java.lang.Object base)</td>
</tr>
<tr>
<td>java.util.Iterator&lt;FeatureDescriptor&gt; getFeatureDescriptors(ELContext context, java.lang.Object base)</td>
</tr>
<tr>
<td>java.lang.Class&lt;?&gt; getType(ELContext context, java.lang.Object base, java.lang.Object property)</td>
</tr>
</tbody>
</table>
Constructors

2.2.3  BeanELResolver()

    public BeanELResolver()

    Creates a new read/write BeanELResolver.

2.2.4  BeanELResolver(boolean)

    public BeanELResolver(boolean isReadOnly)

    Creates a new BeanELResolver whose read-only status is determined by the given parameter.

    Parameters:
    isReadOnly - true if this resolver cannot modify beans; false otherwise.

Methods

2.2.5  getCommonPropertyType(ELContext, Object)

    public java.lang.Class<?>
    getCommonPropertyType(ELContext context, java.lang.Object base)

    If the base object is not null, returns the most general type that this resolver accepts for the property argument. Otherwise, returns null.

    Assuming the base is not null, this method will always return Object.class. This is because any object is accepted as a key and is coerced into a string.

    Overrides: getCommonPropertyType in class ELResolver
2.2.6 `getFeatureDescriptors(ELContext, Object)`

```java
class BeanELResolver {
    public java.util.Iterator<FeatureDescriptor> getFeatureDescriptors(javax.el.ELContext context, java.lang.Object base) {
        if (base != null) {
            return new java.util.Iterator<FeatureDescriptor>() {
                private Iterator<String> iterator = base.getBeanInfo().getPropertyDescriptors().iterator();

                @Override
                public boolean hasNext() {
                    return iterator.hasNext();
                }

                @Override
                public FeatureDescriptor next() {
                    return new FeatureDescriptor(iterator.next());
                }

                @Override
                public void remove() {
                    throw new UnsupportedOperationException();
                }
            };
        } else {
            return null;
        }
    }
}
```

**Parameters:**
- `context` - The context of this evaluation.
- `base` - The bean to analyze.

**Returns:** null if `base` is null; otherwise `Object.class`.

2.2.7 `getType(ELContext, Object, Object)`

```java
class BeanELResolver {
    public java.lang.Class<?> getType(javax.el.ELContext context, java.lang.Object base, java.lang.Object property) {
        if (base != null) {
            propertyResolved = true;
            return base.getClass();
        } else {
            throw new PropertyNotFoundException();
        }
    }
}
```

**Parameters:**
- `context` - The context of this evaluation.
- `base` - The bean to analyze.
- `property` - The name of the property to analyze. Will be coerced to a `String`. 

**Returns:** An `Iterator` containing zero or more `FeatureDescriptor` objects, each representing a property on this bean, or `null` if the `base` object is null.
>Returns: If the `propertyResolved` property of `ELContext` was set to `true`, then the most general acceptable type; otherwise undefined.

**Throws:**
- `java.lang.NullPointerException` - if context is null
- `PropertyNotFoundException` - if base is not null and the specified property does not exist or is not readable.
- `ELException` - if an exception was thrown while performing the property or variable resolution. The thrown exception must be included as the cause property of this exception, if available.

### 2.2.8 `getValue(ELContext, Object, Object)`

```
```

If the base object is not null, returns the current value of the given property on this bean.

If the base is not null, the `propertyResolved` property of the `ELContext` object must be set to `true` by this resolver, before returning. If this property is not `true` after this method is called, the caller should ignore the return value.

The provided property name will first be coerced to a `String`. If the property is a readable property of the base object, as per the JavaBeans specification, then return the result of the getter call. If the getter throws an exception, it is propagated to the caller. If the property is not found or is not readable, a `PropertyNotFoundException` is thrown.

**Overrides:** `getValue` in class `ELResolver`

**Parameters:**
- `context` - The context of this evaluation.
- `base` - The bean on which to get the property.
- `property` - The name of the property to get. Will be coerced to a `String`.

**Returns:** If the `propertyResolved` property of `ELContext` was set to `true`, then the value of the given property. Otherwise, undefined.

**Throws:**
- `java.lang.NullPointerException` - if context is null.
- `PropertyNotFoundException` - if base is not null and the specified property does not exist or is not readable.
- `ELException` - if an exception was thrown while performing the property or variable resolution. The thrown exception must be included as the cause property of this exception, if available.

### 2.2.9 `isReadOnly(ELContext, Object, Object)`

```
public boolean isReadOnly(javax.el.ELContext context, java.lang.Object base, java.lang.Object property)
```

If the base object is not null, returns whether a call to `setValue(ELContext, Object, Object)` will always fail.

If the base is not null, the `propertyResolved` property of the `ELContext` object must be set to `true` by this resolver, before returning. If this property is not `true` after this method is called, the caller can safely assume no value was set.
BeanELResolver \(\text CGRect\) javax.el

setValue(ELContext, Object, Object, Object)

If this resolver was constructed in read-only mode, this method will always return true.

The provided property name will first be coerced to a String. If property is a writable property of base, false is returned. If the property is found but is not writable, true is returned. If the property is not found, a PropertyNotFoundException is thrown.

Overrides: isReadOnly in class ELResolver

Parameters:
- context - The context of this evaluation.
- base - The bean to analyze.
- property - The name of the property to analyzed. Will be coerced to a String.

Returns: If the propertyResolved property of ELContext was set to true, then true if calling the setValue method will always fail or false if it is possible that such a call may succeed; otherwise undefined.

Throws:
- java.lang.NullPointerException - if context is null
- PropertyNotFoundException - if base is not null and the specified property does not exist.
- ELException - if an exception was thrown while performing the property or variable resolution. The thrown exception must be included as the cause property of this exception, if available.

2.2.10 setValue(ELContext, Object, Object, Object)


If the base object is not null, attempts to set the value of the given property on this bean.

If the base is not null, the propertyResolved property of the ELContext object must be set to true by this resolver, before returning. If this property is not true after this method is called, the caller can safely assume no value was set.

If this resolver was constructed in read-only mode, this method will always throw PropertyNotWritableException.

The provided property name will first be coerced to a String. If property is a writable property of base (as per the JavaBeans Specification), the setter method is called (passing value). If the property exists but does not have a setter, then a PropertyNotFoundException is thrown. If the property does not exist, a PropertyNotFoundException is thrown.

Overrides: setValue in class ELResolver

Parameters:
- context - The context of this evaluation.
- base - The bean on which to set the property.
- property - The name of the property to set. Will be coerced to a String.
- val - The value to be associated with the specified key.

Throws:
- java.lang.NullPointerException - if context is null.
- PropertyNotFoundException - if base is not null and the specified property does not exist.
javax.el  BeanELResolver
setValue(ELContext, Object, Object, Object)

PropertyNotWritableException71 - if this resolver was constructed in read-only mode, or if there is no setter for the property.

ELException35 - if an exception was thrown while performing the property or variable resolution. The thrown exception must be included as the cause property of this exception, if available.
2.3  javax.el
BeanELResolver.BeanProperties

2.3.1 Declaration
protected static final class BeanELResolver.BeanProperties
java.lang.Object
   +--javax.el.BeanELResolver.BeanProperties

Enclosing Class: BeanELResolver

Member Summary

Constructors
BeanELResolver.BeanProperties(java.lang.Class baseClass)

Methods
getBeanProperty(java.lang.String property)

Inherited Member Summary

Methods inherited from class Object
clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)

Constructors

2.3.2  BeanELResolver.BeanProperties(Class)
   public  BeanELResolver.BeanProperties(java.lang.Class baseClass)

Methods

2.3.3  getBeanProperty(String)
   public  javax.el.BeanELResolver.BeanProperty getBeanProperty(java.lang.String property)
2.4  javax.el
BeanELResolver.BeanProperty

2.4.1 Declaration
protected static final class BeanELResolver.BeanProperty

java.lang.Object
   +--javax.el.BeanELResolver.BeanProperty

Enclosing Class: BeanELResolver

Member Summary

<table>
<thead>
<tr>
<th>Constructors</th>
<th>BeanELResolver.BeanProperty(java.lang.Class baseClass, java.beans.PropertyDescriptor descriptor)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods</td>
<td>java.lang.Class getPropertyType()</td>
</tr>
<tr>
<td></td>
<td>getReadMethod()</td>
</tr>
<tr>
<td></td>
<td>java.lang.reflect.Method getWriteMethod()</td>
</tr>
<tr>
<td></td>
<td>boolean isReadOnly()</td>
</tr>
</tbody>
</table>

Inherited Member Summary

Methods inherited from class Object
clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)

Constructors

2.4.2 BeanELResolver.BeanProperty(Class, PropertyDescriptor)
    public BeanELResolver.BeanProperty(java.lang.Class baseClass, java.beans.PropertyDescriptor descriptor)
### Methods

2.4.3 `getPropertyType()`

```java
public java.lang.Class getPropertyType()
```

2.4.4 `getReadMethod()`

```java
public java.lang.reflect.Method getReadMethod()
```

2.4.5 `getWriteMethod()`

```java
public java.lang.reflect.Method getWriteMethod()
```

2.4.6 `isReadOnly()`

```java
public boolean isReadOnly()
```
2.5  javax.el

CompositeELResolver

2.5.1  Declaration

```java
public class CompositeELResolver extends ELResolver
```

```java
java.lang.Object
 |  
---javax.el.ELResolver
  
---javax.elCompositeELResolver
```

2.5.2  Description

Maintains an ordered composite list of child ELResolvers.

Though only a single ELResolver is associated with an ELContext, there are usually multiple resolvers considered for any given variable or property resolution. ELResolvers are combined together using a CompositeELResolver, to define rich semantics for evaluating an expression.

For the `getValue(ELContext, Object, Object)`\(^{24}\), `getType(ELContext, Object, Object)`\(^{23}\), `setValue(ELContext, Object, Object, Object)`\(^{26}\) and `isReadOnly(ELContext, Object, Object)`\(^{25}\) methods, an ELResolver is not responsible for resolving all possible (base, property) pairs. In fact, most resolvers will only handle a base of a single type. To indicate that a resolver has successfully resolved a particular (base, property) pair, it must set the `propertyResolved` property of the ELContext to true. If it could not handle the given pair, it must leave this property alone. The caller must ignore the return value of the method if `propertyResolved` is false.

The CompositeELResolver initializes the `ELContext.propertyResolved` flag to false, and uses it as a stop condition for iterating through its component resolvers.

The `ELContext.propertyResolved` flag is not used for the design-time methods `getFeatureDescriptors(ELContext, Object)`\(^{23}\) and `getCommonPropertyType(ELContext, Object)`\(^{22}\). Instead, results are collected and combined from all child ELResolvers for these methods.

Since: JSP 2.1

See Also: `ELContext`\(^{28}\), `ELResolver`\(^{37}\)

### Member Summary

<table>
<thead>
<tr>
<th>Constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CompositeELResolver()</code>(^{22})</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void add(ELResolver elResolver)</code>(^{22})</td>
</tr>
<tr>
<td><code>java.lang.Class&lt;?&gt; getCommonPropertyType(ELContext context, java.lang.Object base)</code>(^{22})</td>
</tr>
<tr>
<td><code>java.util.Iterator&lt;FeatureDescriptor&gt; getFeatureDescriptors(ELContext context, java.lang.Object base)</code>(^{23})</td>
</tr>
</tbody>
</table>
### Member Summary

<table>
<thead>
<tr>
<th>Method Type</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.Class&lt;?&gt;</td>
<td>getType(ELContext context, java.lang.Object base)</td>
<td>Returns the most general type that this resolver accepts for the property argument, given a base object. One use for this method is to assist tools in auto-completion. The result is obtained by querying all component resolvers.</td>
</tr>
<tr>
<td>java.lang.Object</td>
<td>getValue(ELContext context, java.lang.Object base)</td>
<td>Adds the given resolver to the list of component resolvers. Resolvers are consulted in the order in which they are added. Parameters: elResolver - The component resolver to add. Throws: java.lang.NullPointerException - If the provided resolver is null.</td>
</tr>
<tr>
<td>boolean</td>
<td>isReadOnly(ELContext context, java.lang.Object base)</td>
<td></td>
</tr>
</tbody>
</table>
The `Class` returned is the most specific class that is a common superclass of all the classes returned by each component resolver's `getCommonPropertyType` method. If `null` is returned by a resolver, it is skipped.

**Overrides:** `getCommonPropertyType` in class `ELResolver` 39

**Parameters:**
- `context` - The context of this evaluation.
- `base` - The base object to return the most general property type for, or `null` to enumerate the set of top-level variables that this resolver can evaluate.

**Returns:** `null` if this ELResolver does not know how to handle the given `base` object; otherwise `Object.class` if any type of property is accepted; otherwise the most general property type accepted for the given `base`.

### 2.5.6 `getFeatureDescriptors(ELContext, Object)`

```java
public java.util.Iterator<FeatureDescriptor> getFeatureDescriptors(javax.el.ELContext context, java.lang.Object base)
```

Returns information about the set of variables or properties that can be resolved for the given `base` object. One use for this method is to assist tools in auto-completion. The results are collected from all component resolvers.

The `propertyResolved` property of the `ELContext` is not relevant to this method. The results of all `ELResolvers` are concatenated.

The `Iterator` returned is an iterator over the collection of `FeatureDescriptor` objects returned by the iterators returned by each component resolver’s `getFeatureDescriptors` method. If `null` is returned by a resolver, it is skipped.

**Overrides:** `getFeatureDescriptors` in class `ELResolver` 39

**Parameters:**
- `context` - The context of this evaluation.
- `base` - The base object whose set of valid properties is to be enumerated, or `null` to enumerate the set of top-level variables that this resolver can evaluate.

**Returns:** An `Iterator` containing zero or more (possibly infinitely more) `FeatureDescriptor` objects, or `null` if this resolver does not handle the given `base` object or that the results are too complex to represent with this method.

### 2.5.7 `getType(ELContext, Object, Object)`

```java
public java.lang.Class<?> getType(javax.el.ELContext context, java.lang.Object base, java.lang.Object property)
```

For a given `base` and `property`, attempts to identify the most general type that is acceptable for an object to be passed as the value parameter in a future call to the `setValue(ELContext, Object, Object)` method. The result is obtained by querying all component resolvers.

If this resolver handles the given (`base`, `property`) pair, the `propertyResolved` property of the `ELContext` object must be set to `true` by the resolver, before returning. If this property is not `true` after this method is called, the caller should ignore the return value.

First, `propertyResolved` is set to `false` on the provided `ELContext`.

Next, for each component resolver in this composite:
1. The `getType()` method is called, passing in the provided `context`, `base` and `property`.
2. If the `ELContext`'s `propertyResolved` flag is `false` then iteration continues.
3. Otherwise, iteration stops and no more component resolvers are considered. The value returned by `getType()` is returned by this method.

If none of the component resolvers were able to perform this operation, the value `null` is returned and the `propertyResolved` flag remains set to `false`.

Any exception thrown by component resolvers during the iteration is propagated to the caller of this method.

**Overrides:** `getType()` in class `ELResolver`  

**Parameters:**
- `context` - The context of this evaluation.  
- `base` - The base object whose property value is to be analyzed, or `null` to analyze a top-level variable.  
- `property` - The property or variable to return the acceptable type for.

**Returns:** If the `propertyResolved` property of `ELContext` was set to `true`, then the most general acceptable type; otherwise undefined.

**Throws:**  
- `java.lang.NullPointerException` - if `context` is `null`  
- `PropertyNotFoundException` - if the given (base, property) pair is handled by this `ELResolver` but the specified variable or property does not exist or is not readable.  
- `ELException` - if an exception was thrown while performing the property or variable resolution. The thrown exception must be included as the cause property of this exception, if available.

### 2.5.8 `getValue(ELContext, Object, Object)`

```java
```

Attempts to resolve the given `property` object on the given `base` object by querying all component resolvers.

If this resolver handles the given (base, property) pair, the `propertyResolved` property of the `ELContext` object must be set to `true` by the resolver, before returning. If this property is not `true` after this method is called, the caller should ignore the return value.

First, `propertyResolved` is set to `false` on the provided `ELContext`.

Next, for each component resolver in this composite:

1. The `getValue()` method is called, passing in the provided `context`, `base` and `property`.
2. If the `ELContext`'s `propertyResolved` flag is `false` then iteration continues.
3. Otherwise, iteration stops and no more component resolvers are considered. The value returned by `getValue()` is returned by this method.

If none of the component resolvers were able to perform this operation, the value `null` is returned and the `propertyResolved` flag remains set to `false`.
javax.el CompositeELResolver

isReadOnly(ELContext, Object, Object)

Any exception thrown by component resolvers during the iteration is propagated to the caller of this method.

Overrides: getValue in class ELResolver

Parameters:
- context - The context of this evaluation.
- base - The base object whose property value is to be returned, or null to resolve a top-level variable.
- property - The property or variable to be resolved.

Returns: If the propertyResolved property of ELContext was set to true, then the result of the variable or property resolution; otherwise undefined.

Throws:
- java.lang.NullPointerException - if context is null
- PropertyNotFoundException - if the given (base, property) pair is handled by this ELResolver but the specified variable or property does not exist or is not readable.
- ELException - if an exception was thrown while performing the property or variable resolution. The thrown exception must be included as the cause property of this exception, if available.

2.5.9 isReadOnly(ELContext, Object, Object)

public boolean isReadOnly(javax.el.ELContext context, java.lang.Object base, java.lang.Object property)

For a given base and property, attempts to determine whether a call to setValue(ELContext, Object, Object, Object) will always fail. The result is obtained by querying all component resolvers.

If this resolver handles the given (base, property) pair, the propertyResolved property of the ELContext object must be set to true by the resolver, before returning. If this property is not true after this method is called, the caller should ignore the return value.

First, propertyResolved is set to false on the provided ELContext.

Next, for each component resolver in this composite:

1. The isReadOnly() method is called, passing in the provided context, base and property.
2. If the ELContext's propertyResolved flag is false then iteration continues.
3. Otherwise, iteration stops and no more component resolvers are considered. The value returned by isReadOnly() is returned by this method.

If none of the component resolvers were able to perform this operation, the value false is returned and the propertyResolved flag remains set to false.

Any exception thrown by component resolvers during the iteration is propagated to the caller of this method.

Overrides: isReadOnly in class ELResolver

Parameters:
- context - The context of this evaluation.
- base - The base object whose property value is to be analyzed, or null to analyze a top-level variable.
property - The property or variable to return the read-only status for.

Returns: If the propertyResolved property of ELContext was set to true, then true if the property is read-only or false if not; otherwise undefined.

Throws:
java.lang.NullPointerException - if context is null

PropertyNotFoundException - if the given (base, property) pair is handled by this ELResolver but the specified variable or property does not exist.

ELException - if an exception was thrown while performing the property or variable resolution. The thrown exception must be included as the cause property of this exception, if available.

2.5.10 setValue(ELContext, Object, Object, Object)


Attempts to set the value of the given property object on the given base object. All component resolvers are asked to attempt to set the value.

If this resolver handles the given (base, property) pair, the propertyResolved property of the ELContext object must be set to true by the resolver, before returning. If this property is not true after this method is called, the caller can safely assume no value has been set.

First, propertyResolved is set to false on the provided ELContext.

Next, for each component resolver in this composite:

1. The setValue() method is called, passing in the provided context, base, property and value.

2. If the ELContext's propertyResolved flag is false then iteration continues.

3. Otherwise, iteration stops and no more component resolvers are considered.

If none of the component resolvers were able to perform this operation, the propertyResolved flag remains set to false.

Any exception thrown by component resolvers during the iteration is propagated to the caller of this method.

Overrides: setValue in class ELResolver

Parameters:
context - The context of this evaluation.
base - The base object whose property value is to be set, or null to set a top-level variable.
property - The property or variable to be set.
val - The value to set the property or variable to.

Throws:
java.lang.NullPointerException - if context is null

PropertyNotFoundException - if the given (base, property) pair is handled by this ELResolver but the specified variable or property does not exist.

PropertyNotWritableException - if the given (base, property) pair is handled by this ELResolver but the specified variable or property is not writable.
javax.el CompositeELResolver

setValue(ELContext, Object, Object, Object)

ELException - if an exception was thrown while attempting to set the property or variable. The thrown exception must be included as the cause property of this exception, if available.
2.6 javax.el

ELContext

2.6.1 Declaration

public abstract class ELContext

java.lang.Object  
 |  
+--javax.el.ELContext

2.6.2 Description

Context information for expression evaluation.

To evaluate an Expression, an ELContext must be provided. The ELContext holds:

• a reference to the base ELResolver that will be consulted to resolve model objects and their properties

• a reference to FunctionMapper that will be used to resolve EL Functions.

• a reference to VariableMapper that will be used to resolve EL Variables.

• a collection of all the relevant context objects for use by ELResolvers

• state information during the evaluation of an expression, such as whether a property has been resolved yet

The collection of context objects is necessary because each ELResolver may need access to a different context object. For example, JSP and Faces resolvers need access to a javax.servlet.jsp.JspContext and a javax.faces.context.FacesContext, respectively.

Creation of ELContext objects is controlled through the underlying technology. For example, in JSP the JspContext.getELContext() factory method is used. Some technologies provide the ability to add an ELContextListener so that applications and frameworks can ensure their own context objects are attached to any newly created ELContext.

Because it stores state during expression evaluation, an ELContext object is not thread-safe. Care should be taken to never share an ELContext instance between two or more threads.

Since: JSP 2.1

See Also: ELContextListener, ELContextEvent, ELResolver, FunctionMapper, VariableMapper, javax.servlet.jsp.JspContext

Member Summary

Constructors

ELContext()  

Methods

getContext(java.lang.Class key)  
getELResolver()  
getFunctionMapper()  
getLocale()  

28  Expression Language Specification • May 8, 2006
ELContext

### Constructor

2.6.3 **ELContext()**

```java
public ELContext()
```

### Methods

#### 2.6.4 **getContext(Class)**

```java
public java.lang.Object getContext(java.lang.Class key)
```

Returns the context object associated with the given key.

The `ELContext` maintains a collection of context objects relevant to the evaluation of an expression. These context objects are used by `ELResolvers`. This method is used to retrieve the context with the given key from the collection.

By convention, the object returned will be of the type specified by the `key`. However, this is not required and the key is used strictly as a unique identifier.

**Parameters:**

- `key`: The unique identifier that was used to associate the context object with this `ELContext`.

**Returns:** The context object associated with the given key, or null if no such context was found.

**Throws:**

- `java.lang.NullPointerException` - if `key` is null.

#### 2.6.5 **getELResolver()**

```java
public abstract javax.el.ELResolver getELResolver()
```

Retrieves the `ELResolver` associated with this context.
The ELContext maintains a reference to the ELResolver that will be consulted to resolve variables and properties during an expression evaluation. This method retrieves the reference to the resolver.

Once an ELContext is constructed, the reference to the ELResolver associated with the context cannot be changed.

Returns: The resolver to be consulted for variable and property resolution during expression evaluation.

### 2.6.6 getFunctionMapper()

```java
public abstract javax.el.FunctionMapper getFunctionMapper()
```

Retrieves the FunctionMapper associated with this ELContext.

Returns: The function mapper to be consulted for the resolution of EL functions.

### 2.6.7 getLocale()

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

Get the Locale stored by a previous invocation to `setLocale(Locale)`. If this method returns non null, this Locale must be used for all localization needs in the implementation. The Locale must not be cached to allow for applications that change Locale dynamically.

Returns: The Locale in which this instance is operating. Used primarily for message localization.

### 2.6.8 getVariableMapper()

```java
public abstract javax.el.VariableMapper getVariableMapper()
```

Retrieves the VariableMapper associated with this ELContext.

Returns: The variable mapper to be consulted for the resolution of EL variables.

### 2.6.9 isPropertyResolved()

```java
public boolean isPropertyResolved()
```

Returns whether an ELResolver has successfully resolved a given (base, property) pair.

The CompositeELResolver checks this property to determine whether it should consider or skip other component resolvers.

Returns: true if the property has been resolved, or false if not.

See Also: CompositeELResolver

### 2.6.10 putContext(Class, Object)

```java
public void putContext(java.lang.Class key, java.lang.Object contextObject)
```

Associates a context object with this ELContext.

The ELContext maintains a collection of context objects relevant to the evaluation of an expression. These context objects are used by ELResolvers. This method is used to add a context object to that collection.

By convention, the contextObject will be of the type specified by the key. However, this is not required and the key is used strictly as a unique identifier.

Parameters:
- key - The key used by an ELResolver to identify this context object.
contextObject - The context object to add to the collection.

Throws:
java.lang.NullPointerException - if key is null or contextObject is null.

2.6.11 setLocale(Locale)

public void setLocale(java.util.Locale locale)

Set the Locale for this instance. This method may be called by the party creating the instance, such as JavaServer Faces or JSP, to enable the EL implementation to provide localized messages to the user. If no Locale is set, the implementation must use the locale returned by Locale.getDefault().

2.6.12 setPropertyResolved(boolean)

public void setPropertyResolved(boolean resolved)

Called to indicate that a ELResolver has successfully resolved a given (base, property) pair.

The CompositeELResolver checks this property to determine whether it should consider or skip other component resolvers.

Parameters:
resolved - true if the property has been resolved, or false if not.

See Also: CompositeELResolver
2.7 javax.el

ELContextEvent

2.7.1 Declaration

```java
public class ELContextEvent extends java.util.EventObject

java.lang.Object
    |---java.util.EventObject
        |---javax.el.ELContextEvent
```

All Implemented Interfaces: java.io.Serializable

2.7.2 Description

An event which indicates that an ELContext has been created. The source object is the ELContext that was created.

Since: JSP 2.1

See Also: ELContext, ELContextListener

---

### Member Summary

<table>
<thead>
<tr>
<th>Constructors</th>
<th>ELContextEvent(ELContext source)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
<th>ELContext getELContext()</th>
</tr>
</thead>
</table>

---

### Inherited Member Summary

Fields inherited from class EventObject

source

Methods inherited from class EventObject

getSource(), toString()

Methods inherited from class Object

clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), wait(), wait(long), wait(long, int)
Constructors

2.7.3 ELContextEvent(ELContext)

public ELContextEvent(javax.el.ELContext source)

Constructs an ELContextEvent object to indicate that an ELContext has been created.

Parameters:

source - the ELContext that was created.

Methods

2.7.4 getELContext()

public javax.el.ELContext getELContext()

Returns the ELContext that was created. This is a type-safe equivalent of the java.util.EventObject.getSource() method.

Returns: the ELContext that was created.
2.8 javax.el ELContextListener

2.8.1 Declaration
public interface ELContextListener extends java.util.EventListener

All Superinterfaces: java.util.EventListener

2.8.2 Description
The listener interface for receiving notification when an ELContext is created.

Since: JSP 2.1

See Also: ELContext, ELContextEvent

Member Summary

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>void contextCreated(ELContextEvent ece)</td>
</tr>
</tbody>
</table>

Methods

2.8.3 contextCreated(ELContextEvent)

public void contextCreated(javax.el.ELContextEvent ece)

Invoked when a new ELContext has been created.

Parameters:

ece - the notification event.
2.9  javax.el

ELException

2.9.1 Declaration

public class ELException extends java.lang.RuntimeException

java.lang.Object
  +--java.lang.Throwable
     +--java.lang.Exception
        +--java.lang.RuntimeException
            +--javax.el.ELException

All Implemented Interfaces: java.io.Serializable

Direct Known Subclasses: MethodNotFoundException, PropertyNotFoundException, PropertyNotWritableException

2.9.2 Description

Represents any of the exception conditions that can arise during expression evaluation.

Since: JSP 2.1

Member Summary

Constructors

ELException()  
ELException(java.lang.String pMessage)  
ELException(java.lang.String pMessage, java.lang.Throwable pRootCause)  
ELException(java.lang.Throwables pRootCause)

Inherited Member Summary

Methods inherited from class Object

clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), wait(), wait(long), wait(long, int)

Methods inherited from class Throwable

fillInStackTrace(), getCause(), getLocalizedMessage(), getMessage(), getStackTrace(), initCause(Throwable), printStackTrace(), printStackTrace(PrintStream), printStackTrace(PrintWriter), setStackTrace(StackTraceElement[]), toString()
Constructors

2.9.3 ELException()

    public ELException()

    Creates an ELException with no detail message.

2.9.4 ELException(String)

    public ELException(java.lang.String pMessage)

    Creates an ELException with the provided detail message.

    Parameters:
    pMessage - the detail message

2.9.5 ELException(Throwable)

    public ELException(java.lang.Throwable pRootCause)

    Creates an ELException with the given cause.

    Parameters:
    pRootCause - the originating cause of this exception

2.9.6 ELException(String, Throwable)

    public ELException(java.lang.String pMessage, java.lang.Throwable pRootCause)

    Creates an ELException with the given detail message and root cause.

    Parameters:
    pMessage - the detail message
    pRootCause - the originating cause of this exception
2.10  javax.el

ELResolver

2.10.1  Declaration

public abstract class ELResolver

java.lang.Object

|--javax.el.ELResolver

Direct Known Subclasses:  ArrayELResolver, BeanELResolver,
                          CompositeELResolver, ListELResolver,
                          MapELResolver,
                          ResourceBundleELResolver

2.10.2  Description

Enables customization of variable and property resolution behavior for EL expression evaluation.

While evaluating an expression, the ELResolver associated with the ELContext is consulted to do the
initial resolution of the first variable of an expression. It is also consulted when a . or [] operator is
encountered, except for the last such operator in a method expression, in which case the resolution rules are hard
coded.

For example, in the EL expression ${employee.lastName}, the ELResolver determines what object
employee refers to, and what it means to get the lastName property on that object.

Most methods in this class accept a base and property parameter. In the case of variable resolution (e.g.
determining what employee refers to in ${employee.lastName}), the base parameter will be null
and the property parameter will always be of type String. In this case, if the property is not a String,
the behavior of the ELResolver is undefined.

In the case of property resolution, the base parameter identifies the base object and the property object
identifies the property on that base. For example, in the expression ${employee.lastName}.base is the
result of the variable resolution for employee and property is the string “lastName”. In the expression
${y[x]}, base is the result of the variable resolution for y and property is the result of the variable
resolution for x.

Though only a single ELResolver is associated with an ELContext, there are usually multiple resolvers
considered for any given variable or property resolution. ELResolvers are combined together using
CompositeELResolver to define rich semantics for evaluating an expression.

For the getValue(ELContext, Object, Object), getType(ELContext, Object, Object),
setValue(ELContext, Object, Object, Object) and
isReadOnly(ELContext, Object, Object) methods, an ELResolver is not responsible for
resolving all possible (base, property) pairs. In fact, most resolvers will only handle a base of a single type. To
indicate that a resolver has successfully resolved a particular (base, property) pair, it must set the
propertyResolved property of the ELContext to true. If it could not handle the given pair, it must
leave this property alone. The caller must ignore the return value of the method if propertyResolved is
false.

The getFeatureDescriptors(ELContext, Object) and
getCommonPropertyType(ELContext, Object) methods are primarily designed for design-time
tool support, but must handle invocation at runtime as well. The java.beans.Beans.isDesignTime() method can be used to determine if the resolver is being consulted at design-time or runtime.
Since: JSP 2.1

See Also: CompositeELResolver, ELContext.getELResolver()

Member Summary

Fields

<table>
<thead>
<tr>
<th>static java.lang.String</th>
<th>RESOLVABLE_AT_DESIGN_TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>static java.lang.String</td>
<td>TYPE</td>
</tr>
</tbody>
</table>

Constructors

| ELResolver() |

Methods

<table>
<thead>
<tr>
<th>abstract java.lang.Class&lt;?&gt;</th>
<th>getCommonPropertyType(ELContext context, java.lang.Object base)</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract java.util.Iterator&lt;FeatureDescriptor&gt;</td>
<td>getFeatureDescriptors(ELContext context, java.lang.Object base)</td>
</tr>
<tr>
<td>abstract java.lang.Class&lt;?&gt;</td>
<td>getType(ELContext context, java.lang.Object base, java.lang.Object property)</td>
</tr>
<tr>
<td>abstract boolean</td>
<td>isReadOnly(ELContext context, java.lang.Object base, java.lang.Object property)</td>
</tr>
</tbody>
</table>

Inherited Member Summary

Methods inherited from class Object

| clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int) |

Fields

2.10.3 RESOLVABLE_AT_DESIGN_TIME

public static final java.lang.String RESOLVABLE_AT_DESIGN_TIME

The attribute name of the named attribute in the FeatureDescriptor that specifies whether the variable or property can be resolved at runtime.

2.10.4 TYPE

public static final java.lang.String TYPE
The attribute name of the named attribute in the FeatureDescriptor that specifies the runtime type of the variable or property.

### Constructors

**2.10.5 ELResolver()**

```java
public ELResolver()
```

### Methods

**2.10.6 getCommonPropertyType(ELContext, Object)**

```java
public abstract java.lang.Class<?>
getCommonPropertyType(javax.el.ELContext context,
java.lang.Object base)
```

Returns the most general type that this resolver accepts for the property argument, given a base object. One use for this method is to assist tools in auto-completion. This assists tools in auto-completion and also provides a way to express that the resolver accepts a primitive value, such as an integer index into an array. For example, the ArrayELResolver will accept any int as a property, so the return value would be Integer.class.

**Parameters:**
- `context`: The context of this evaluation.
- `base`: The base object to return the most general property type for, or null to enumerate the set of top-level variables that this resolver can evaluate.

**Returns:** null if this ELResolver does not know how to handle the given base object; otherwise Object.class if any type of property is accepted; otherwise the most general property type accepted for the given base.

**2.10.7 getFeatureDescriptors(ELContext, Object)**

```java
public abstract java.util.Iterator<FeatureDescriptor>
getFeatureDescriptors(javax.el.ELContext context,
java.lang.Object base)
```

Returns information about the set of variables or properties that can be resolved for the given base object. One use for this method is to assist tools in auto-completion. If the base parameter is null, the resolver must enumerate the list of top-level variables it can resolve. The Iterator returned must contain zero or more instances of java.beans.FeatureDescriptor, in no guaranteed order. In the case of primitive types such as int, the value null must be returned. This is to prevent the useless iteration through all possible primitive values. A return value of null indicates that this resolver does not handle the given base object or that the results are too complex to represent with this method and the getCommonPropertyType(ELContext, Object) method should be used instead.

Each FeatureDescriptor will contain information about a single variable or property. In addition to the standard properties, the FeatureDescriptor must have two named attributes (as set by the setValue method):

- **TYPE**: The value of this named attribute must be an instance of java.lang.Class and specify
the runtime type of the variable or property.

- **RESOLVABLE_AT_DESIGN_TIME**: The value of this named attribute must be an instance of `java.lang.Boolean` and indicates whether it is safe to attempt to resolve this property at design-time. For instance, it may be unsafe to attempt a resolution at design time if the `ELResolver` needs access to a resource that is only available at runtime and no acceptable simulated value can be provided.

The caller should be aware that the `Iterator` returned might iterate through a very large or even infinitely large set of properties. Care should be taken by the caller to not get stuck in an infinite loop.

This is a “best-effort” list. Not all `ELResolvers` will return completely accurate results, but all must be callable at both design-time and runtime (i.e. whether or not `Beans.isDesignTime()` returns `true`), without causing errors.

The `propertyResolved` property of the `ELContext` is not relevant to this method. The results of all `ELResolvers` are concatenated in the case of composite resolvers.

**Parameters:**
- `context` - The context of this evaluation.
- `base` - The base object whose set of valid properties is to be enumerated, or `null` to enumerate the set of top-level variables that this resolver can evaluate.

**Returns:** An `Iterator` containing zero or more (possibly infinitely more) `FeatureDescriptor` objects, or `null` if this resolver does not handle the given `base` object or that the results are too complex to represent with this method.

**See Also:** `java.beans.FeatureDescriptor`
javax.el

ELResolver

g getValue(ELContext, Object, Object)

PropertyNotFoundExceptionprocessed - if the given (base, property) pair is handled by this ELResolver but the specified variable or property does not exist or is not readable.

ELExceptionprocessed - if an exception was thrown while performing the property or variable resolution. The thrown exception must be included as the cause property of this exception, if available.

2.10.9 getValue(ELContext, Object, Object)

g public abstract java.lang.Object getValue(javax.el.ELContext context, java.lang.Object base, java.lang.Object property)

Attempts to resolve the given property object on the given base object.

If this resolver handles the given (base, property) pair, the propertyResolved property of the ELContext object must be set to true by the resolver, before returning. If this property is not true after this method is called, the caller should ignore the return value.

Parameters:
context - The context of this evaluation.
base - The base object whose property value is to be returned, or null to resolve a top-level variable.
property - The property or variable to be resolved.

Returns: If the propertyResolved property of ELContext was set to true, then the result of the variable or property resolution; otherwise undefined.

Throws:
java.lang.NullPointerException - if context is null
PropertyNotFoundExceptionprocessed - if the given (base, property) pair is handled by this ELResolver but the specified variable or property does not exist or is not readable.
ELExceptionprocessed - if an exception was thrown while performing the property or variable resolution.

2.10.10 isReadOnly(ELContext, Object, Object)

g public abstract boolean isReadOnly(javax.el.ELContext context, java.lang.Object base, java.lang.Object property)

For a given base and property, attempts to determine whether a call to setValue(ELContext, Object, Object, Object) will always fail.

If this resolver handles the given (base, property) pair, the propertyResolved property of the ELContext object must be set to true by the resolver, before returning. If this property is not true after this method is called, the caller should ignore the return value.

Parameters:
context - The context of this evaluation.
base - The base object whose property value is to be analyzed, or null to analyze a top-level variable.
property - The property or variable to return the read-only status for.

Returns: If the propertyResolved property of ELContext was set to true, then true if the property is read-only or false if not; otherwise undefined.

Throws:
java.lang.NullPointerException - if context is null
ELResolver javax.el
setValue(ELContext, Object, Object, Object)

- PropertyNotFoundException - if the given (base, property) pair is handled by this ELResolver but the specified variable or property does not exist.
- ELException - if an exception was thrown while performing the property or variable resolution. The thrown exception must be included as the cause property of this exception, if available.

2.10.11 setValue(ELContext, Object, Object, Object)


Attempts to set the value of the given property object on the given base object.

If this resolver handles the given (base, property) pair, the propertyResolved property of the ELContext object must be set to true by the resolver, before returning. If this property is not true after this method is called, the caller can safely assume no value has been set.

Parameters:
- context - The context of this evaluation.
- base - The base object whose property value is to be set, or null to set a top-level variable.
- property - The property or variable to be set.
- value - The value to set the property or variable to.

Throws:
- java.lang.NullPointerException - if context is null
- PropertyNotFoundException - if the given (base, property) pair is handled by this ELResolver but the specified variable or property does not exist.
- PropertyNotWritableException - if the given (base, property) pair is handled by this ELResolver but the specified variable or property is not writable.
- ELException - if an exception was thrown while attempting to set the property or variable. The thrown exception must be included as the cause property of this exception, if available.
2.11 javax.el

**Expression**

2.11.1 Declaration

```java
public abstract class Expression implements java.io.Serializable
```

```
java.lang.Object
+-- javax.el.Expression
```

**All Implemented Interfaces:** java.io.Serializable

**Direct Known Subclasses:** MethodExpression, ValueExpression

2.11.2 Description

Base class for the expression subclasses ValueExpression and MethodExpression, implementing characteristics common to both.

All expressions must implement the `equals()` and `hashCode()` methods so that two expressions can be compared for equality. They are redefined abstract in this class to force their implementation in subclasses.

All expressions must also be `Serializable` so that they can be saved and restored.

Expressions are also designed to be immutable so that only one instance needs to be created for any given expression String / FunctionMapper. This allows a container to pre-create expressions and not have to re-parse them each time they are evaluated.

**Since:** JSP 2.1

<table>
<thead>
<tr>
<th>Member Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Constructors</strong></td>
</tr>
<tr>
<td>Expression()</td>
</tr>
</tbody>
</table>

| **Methods** |
| abstract boolean equals(java.lang.Object obj) |
| abstract java.lang.String getExpressionString() |
| abstract int hashCode() |
| abstract boolean isLiteralText() |

| Inherited Member Summary |
| **Methods inherited from class Object** |
| clone(), finalize(), getClass(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int) |
Constructors

2.11.3 Expression()

    public Expression()

Methods

2.11.4 equals(Object)

    public abstract boolean equals(java.lang.Object obj)

    Determines whether the specified object is equal to this Expression.
    The result is true if and only if the argument is not null, is an Expression object that is the of the
    same type (ValueExpression or MethodExpression), and has an identical parsed representation.
    Note that two expressions can be equal if their expression Strings are different. For example,
    ${fn1:foo()} and ${fn2:foo()} are equal if their corresponding FunctionMappers mapped
    fn1:foo and fn2:foo to the same method.

    Overrides: equals in class Object

    Parameters:
    obj - the Object to test for equality.

    Returns: true if obj equals this Expression; false otherwise.

    See Also: java.util.Hashtable, java.lang.Object.equals(Object)

2.11.5 getExpressionString()

    public abstract java.lang.String getExpressionString()

    Returns the original String used to create this Expression, unmodified.

    This is used for debugging purposes but also for the purposes of comparison (e.g. to ensure the expression
    in a configuration file has not changed).

    This method does not provide sufficient information to re-create an expression. Two different expressions
    can have exactly the same expression string but different function mappings. Serialization should be used to
    save and restore the state of an Expression.

    Returns: The original expression String.

2.11.6 hashCode()

    public abstract int hashCode()

    Returns the hash code for this Expression.

    See the note in the equals(Object) method on how two expressions can be equal if their expression
    Strings are different. Recall that if two objects are equal according to the equals(Object) method, then
calling the hashCode method on each of the two objects must produce the same integer result.

    Implementations must take special note and implement hashCode correctly.

    Overrides: hashCode in class Object
Returns: The hash code for this Expression.

See Also: equals(Object), java.util.Hashtable, java.lang.Object.hashCode()

2.11.7 isLiteralText()

public abstract boolean isLiteralText()

Returns whether this expression was created from only literal text.

This method must return true if and only if the expression string this expression was created from contained no unescaped EL delimiters (${...} or #{...}).

Returns: true if this expression was created from only literal text; false otherwise.
ExpressionFactory

2.12  javax.el
ExpressionFactory

2.12.1 Declaration
public abstract class ExpressionFactory

java.lang.Object
| +--javax.el.ExpressionFactory

2.12.2 Description
Parses a String into a ValueExpression or MethodExpression instance for later evaluation.

Classes that implement the EL expression language expose their functionality via this abstract class. There is no concrete implementation of this API available in this package. Technologies such as JavaServer Pages and JavaServer Faces provide access to an implementation via factory methods.

The createValueExpression(ELContext, String, Class) method is used to parse expressions that evaluate to values (both l-values and r-values are supported). The createMethodExpression(ELContext, String, Class, Class[]) method is used to parse expressions that evaluate to a reference to a method on an object.

Unlike previous incarnations of this API, there is no way to parse and evaluate an expression in one single step. The expression needs to first be parsed, and then evaluated.

Resolution of model objects is performed at evaluation time, via the ELResolver associated with the ELContext passed to the ValueExpression or MethodExpression.

The ELContext object also provides access to the FunctionMapper and VariableMapper to be used when parsing the expression. EL function and variable mapping is performed at parse-time, and the results are bound to the expression. Therefore, the ELContext, FunctionMapper, and VariableMapper are not stored for future use and do not have to be Serializable.

The createValueExpression and createMethodExpression methods must be thread-safe. That is, multiple threads may call these methods on the same ExpressionFactory object simultaneously. Implementations should synchronize access if they depend on transient state. Implementations should not, however, assume that only one object of each ExpressionFactory type will be instantiated; global caching should therefore be static.

The ExpressionFactory must be able to handle the following types of input for the expression parameter:
• Single expressions using the ${} delimiter (e.g. "${employee.lastName}").
• Single expressions using the #{} delimiter (e.g. "#{employee.lastName}").
• Literal text containing no ${} or #{} delimiters (e.g. “John Doe”).
• Multiple expressions using the same delimiter (e.g. "${employee.firstName}${employee.lastName}" or 
  
  
  "#{employee.firstName}#{employee.lastName}").
• Mixed literal text and expressions using the same delimiter (e.g. “Name: ${employee.firstName} 
  
  
  ${employee.lastName}”.

The following types of input are illegal and must cause an ELException to be thrown:
• Multiple expressions using different delimiters (e.g. "$\{employee.firstName}#{employee.lastName}\$").

• Mixed literal text and expressions using different delimiters (e.g. "Name: $\{employee.firstName\} #{employee.lastName}\$").

Since: JSP 2.1

Member Summary

<table>
<thead>
<tr>
<th>Constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ExpressionFactory()</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract <strong>coerceToType</strong>(java.lang.Object <code>obj</code>, java.lang.Class&lt;?&gt; <code>targetType</code>)</td>
</tr>
<tr>
<td>abstract <strong>createMethodExpression</strong>(ELContext <code>context</code>, java.lang.String <code>expression</code>, java.lang.Class&lt;?&gt; `expectedReturnType`, java.lang.Class&lt;?&gt;[] <code>expectedParamTypes</code>)</td>
</tr>
<tr>
<td>abstract <strong>createValueExpression</strong>(ELContext <code>context</code>, java.lang.String <code>expression</code>, java.lang.Class&lt;?&gt; <code>expectedType</code>)</td>
</tr>
<tr>
<td>abstract <strong>createValueExpression</strong>(java.lang.Object <code>instance</code>, java.lang.Class&lt;?&gt; <code>expectedType</code>)</td>
</tr>
</tbody>
</table>

Inherited Member Summary

<table>
<thead>
<tr>
<th>Methods inherited from class Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)</td>
</tr>
</tbody>
</table>

Constructors

2.12.3 **ExpressionFactory()**

```
public ExpressionFactory()
```

Methods

2.12.4 **coerceToType**(Object, Class<?>)

```
public abstract java.lang.Object coerceToType(java.lang.Object `obj`, java.lang.Class<?> `targetType`)
```

Coerces an object to a specific type according to the EL type conversion rules.
An ELException is thrown if an error results from applying the conversion rules.
Parameters:

obj - The object to coerce.

targetType - The target type for the coercion.

Throws:

ELException - thrown if an error results from applying the conversion rules.

2.12.5 `createMethodExpression(ELContext, String, Class<?>, Class<?>[]])`

```java
public abstract javax.el.MethodExpression createMethodExpression(ELContext context, String expression, Class<?> expectedReturnType, Class<?>[] expectedParamTypes)
```

Parses an expression into a `MethodExpression` for later evaluation. Use this method for expressions that refer to methods.

If the expression is a String literal, a `MethodExpression` is created, which when invoked, returns the String literal, coerced to expectedReturnType. An ELException is thrown if expectedReturnType is void or if the coercion of the String literal to the expectedReturnType yields an error (see Section “1.16 Type Conversion”).

This method should perform syntactic validation of the expression. If in doing so it detects errors, it should raise an ELException.

Parameters:

- context - The EL context used to parse the expression. The FunctionMapper and VariableMapper stored in the ELContext are used to resolve functions and variables found in the expression. They can be `null`, in which case functions or variables are not supported for this expression. The object returned must invoke the same functions and access the same variable mappings regardless of whether the mappings in the provided FunctionMapper and VariableMapper instances change between calling `ExpressionFactory.createMethodExpression()` and any method on `MethodExpression`.

- expression - The expression to parse

- expectedReturnType - The expected return type for the method to be found. After evaluating the expression, the MethodExpression must check that the return type of the actual method matches this type. Passing in a value of `null` indicates the caller does not care what the return type is, and the check is disabled.

- expectedParamTypes - The expected parameter types for the method to be found. Must be an array with no elements if there are no parameters expected. It is illegal to pass `null`.

Returns: The parsed expression

Throws:

ELException - Thrown if there are syntactical errors in the provided expression.

java.lang.NullPointerException - if paramTypes is `null`.

2.12.6 `createValueExpression(ELContext, String, Class<?>)`

```java
public abstract javax.el.ValueExpression createValueExpression(ELContext context, String expression, Class<?> expectedType)
```


Parses an expression into a `ValueExpression` for later evaluation. Use this method for expressions that refer to values.

This method should perform syntactic validation of the expression. If in doing so it detects errors, it should raise an `ELException`.

**Parameters:**
- `context` - The EL context used to parse the expression. The `FunctionMapper` and `VariableMapper` stored in the ELContext are used to resolve functions and variables found in the expression. They can be null, in which case functions or variables are not supported for this expression. The object returned must invoke the same functions and access the same variable mappings regardless of whether the mappings in the provided `FunctionMapper` and `VariableMapper` instances change between calling `ExpressionFactory.createValueExpression()` and any method on `ValueExpression`.

  Note that within the EL, the ${} and #{} syntaxes are treated identically. This includes the use of `VariableMapper` and `FunctionMapper` at expression creation time. Each is invoked if not null, independent of whether the #{} or ${} syntax is used for the expression.

- `expression` - The expression to parse

- `expectedType` - The type the result of the expression will be coerced to after evaluation

**Returns:** The parsed expression

**Throws:**
- `java.lang.NullPointerException` - Thrown if `expectedType` is null.
- `ELException` - Thrown if there are syntactical errors in the provided expression.

### 2.12.7 `createValueExpression(Object, Class<?>)`

```java
public abstract javax.el.ValueExpression createValueExpression(java.lang.Object instance, java.lang.Class<?> expectedType)
```

Creates a `ValueExpression` that wraps an object instance. This method can be used to pass any object as a `ValueExpression`. The wrapper `ValueExpression` is read only, and returns the wrapped object via its `getValue()` method, optionally coerced.

**Parameters:**
- `instance` - The object instance to be wrapped.

- `expectedType` - The type the result of the expression will be coerced to after evaluation. There will be no coercion if it is `Object.class`,
2.13 javax.el

FunctionMapper

2.13.1 Declaration
public abstract class FunctionMapper

java.lang.Object
| +--javax.el.FunctionMapper

2.13.2 Description
The interface to a map between EL function names and methods.
A FunctionMapper maps ${prefix:name()} style functions to a static method that can execute that function.

Since: JSP 2.1

Member Summary

Constructors

FunctionMapper()

Methods

abstract resolveFunction(java.lang.String prefix, java.lang.String localName)

Inherited Member Summary

Methods inherited from class Object

clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)

Constructors

2.13.3 FunctionMapper()

public FunctionMapper()
Methods

2.13.4 resolveFunction(String, String)

    public abstract java.lang.reflect.Method resolveFunction(java.lang.String prefix,
            java.lang.String localName)

Resolves the specified prefix and local name into a java.lang.Method.

Returns null if no function could be found that matches the given prefix and local name.

Parameters:
    prefix - the prefix of the function, or "" if no prefix. For example, "fn" in \${fn:method()}, or "" in \${method()}.  
    localName - the short name of the function. For example, "method" in \${fn:method()}.  

Returns: the static method to invoke, or null if no match was found.
2.14 javax.el
ListELResolver

2.14.1 Declaration

public class ListELResolver extends ELResolver

java.lang.Object
  |--javax.el.ELResolver
  |   |--javax.el.ListELResolver

2.14.2 Description

Defines property resolution behavior on instances of java.util.List.

This resolver handles base objects of type java.util.List. It accepts any object as a property and coerces that object into an integer index into the list. The resulting value is the value in the list at that index.

This resolver can be constructed in read-only mode, which means that isReadOnly will always return true and setValue(ELContext, Object, Object, Object) will always throw PropertyNotWritableException.

ELResolvers are combined together using CompositeELResolver, to define rich semantics for evaluating an expression. See the javadocs for ELResolver for details.

Since: JSP 2.1

See Also: CompositeELResolver, ELResolver, java.util.List

Member Summary

<table>
<thead>
<tr>
<th>Constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListELResolver()</td>
</tr>
<tr>
<td>ListELResolver(boolean isReadOnly)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.Class&lt;?&gt; getCommonPropertyType(ELContext context, java.lang.Object base)</td>
</tr>
<tr>
<td>java.util.Iterator&lt;FeatureDescriptor&gt; getFeatureDescriptors(ELContext context, java.lang.Object base)</td>
</tr>
<tr>
<td>java.lang.Class&lt;?&gt; getType(ELContext context, java.lang.Object base, java.lang.Object property)</td>
</tr>
<tr>
<td>java.lang.Object getValue(ELContext context, java.lang.Object base, java.lang.Object property)</td>
</tr>
<tr>
<td>boolean isReadOnly(ELContext context, java.lang.Object base, java.lang.Object property)</td>
</tr>
</tbody>
</table>
Inherited Member Summary

Fields inherited from class [ELResolver]
RESOLVABLE_AT_DESIGN_TIME, TYPE

Methods inherited from class [Object]
clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)

Constructors

2.14.3 ListELResolver()

public [ListELResolver]

Creates a new read/write ListELResolver.

2.14.4 ListELResolver(boolean)

public [ListELResolver](boolean isReadOnly)

Creates a new ListELResolver whose read-only status is determined by the given parameter.

Parameters:

isReadOnly - true if this resolver cannot modify lists; false otherwise.

Methods

2.14.5 getCommonPropertyType(ELContext, Object)

public java.lang.Class<?> getCommonPropertyType(javax.el.ELContext context, java.lang.Object base)

If the base object is a list, returns the most general type that this resolver accepts for the property argument. Otherwise, returns null.

Assuming the base is a List, this method will always return Integer.class. This is because Lists accept integers as their index.

Overrides: getCommonPropertyType in class [ELResolver]

Parameters:

context - The context of this evaluation.

base - The list to analyze. Only bases of type List are handled by this resolver.

Returns: null if base is not a List; otherwise Integer.class.

2.14.6 getFeatureDescriptors(ELContext, Object)

public java.util.Iterator<FeatureDescriptor> getFeatureDescriptors(javax.el.ELContext context, java.lang.Object base)

Always returns null, since there is no reason to iterate through set set of all integers.
The `getCommonPropertyType(ELContext, Object)` method returns sufficient information about what properties this resolver accepts.

**Overrides:** `getFeatureDescriptors` in class `ELResolver`

**Parameters:**
- `context` - The context of this evaluation.
- `base` - The list. Only bases of type `List` are handled by this resolver.

**Returns:** `null`.

### 2.14.7 `getType(ELContext, Object, Object)`

```
public java.lang.Class<?> getType(javax.el.ELContext context, java.lang.Object base, java.lang.Object property)
```

If the base object is a list, returns the most general acceptable type for a value in this list.

If the base is a `List`, the `propertyResolved` property of the `ELContext` object must be set to `true` by this resolver, before returning. If this property is not `true` after this method is called, the caller should ignore the return value.

Assuming the base is a `List`, this method will always return `Object.class`. This is because `Lists` accept any object as an element.

**Overrides:** `getType` in class `ELResolver`

**Parameters:**
- `context` - The context of this evaluation.
- `base` - The list to analyze. Only bases of type `List` are handled by this resolver.
- `property` - The index of the element in the list to return the acceptable type for. Will be coerced into an integer, but otherwise ignored by this resolver.

**Returns:** If the `propertyResolved` property of `ELContext` was set to `true`, then the most general acceptable type; otherwise undefined.

**Throws:**
- `PropertyNotFoundException` - if the given index is out of bounds for this list.
- `java.lang.NullPointerException` - if `context` is `null`
- `ELException` - if an exception was thrown while performing the property or variable resolution. The thrown exception must be included as the cause property of this exception, if available.

### 2.14.8 `getValue(ELContext, Object, Object)`

```
```

If the base object is a list, returns the value at the given index. The index is specified by the `property` argument, and coerced into an integer. If the coercion could not be performed, an `IllegalArgumentException` is thrown. If the index is out of bounds, `null` is returned.

If the base is a `List`, the `propertyResolved` property of the `ELContext` object must be set to `true` by this resolver, before returning. If this property is not `true` after this method is called, the caller should ignore the return value.

**Overrides:** `getValue` in class `ELResolver`
javax.el
ListELResolver

isReadOnly(ELContext, Object, Object)

Parameters:
  context  - The context of this evaluation.
  base  - The list to be analyzed. Only bases of type List are handled by this resolver.
  property  - The index of the value to be returned. Will be coerced into an integer.

Returns: If the propertyResolved property of ELContext was set to true, then the value at the given index or null if the index was out of bounds. Otherwise, undefined.

Throws:
  java.lang.IllegalArgumentException  - if the property could not be coerced into an integer.
  java.lang.NullPointerException  - if context is null.
  ELException  - if an exception was thrown while performing the property or variable resolution. The thrown exception must be included as the cause property of this exception, if available.

2.14.9 isReadOnly(ELContext, Object, Object)

public boolean isReadOnly(javax.el.ELContext context, java.lang.Object base, java.lang.Object property)

If the base object is a list, returns whether a call to setValue(ELContext, Object, Object, Object) will always fail.

If the base is a List, the propertyResolved property of the ELContext object must be set to true by this resolver, before returning. If this property is not true after this method is called, the caller should ignore the return value.

If this resolver was constructed in read-only mode, this method will always return true.

If a List was created using java.util.Collections.unmodifiableList(List), this method must return true. Unfortunately, there is no Collections API method to detect this. However, an implementation can create a prototype unmodifiable List and query its runtime type to see if it matches the runtime type of the base object as a workaround.

Overrides: isReadOnly in class ELResolver

Parameters:
  context  - The context of this evaluation.
  base  - The list to analyze. Only bases of type List are handled by this resolver.
  property  - The index of the element in the list to return the acceptable type for. Will be coerced into an integer, but otherwise ignored by this resolver.

Returns: If the propertyResolved property of ELContext was set to true, then true if calling the setValue method will always fail or false if it is possible that such a call may succeed; otherwise undefined.

Throws:
  PropertyNotFoundException  - if the given index is out of bounds for this list.
  java.lang.NullPointerException  - if context is null
  ELException  - if an exception was thrown while performing the property or variable resolution. The thrown exception must be included as the cause property of this exception, if available.
2.14.10  setValue(ELContext, Object, Object, Object)


If the base object is a list, attempts to set the value at the given index with the given value. The index is specified by the property argument, and coerced into an integer. If the coercion could not be performed, an IllegalArgumentException is thrown. If the index is out of bounds, a PropertyNotFoundException is thrown.

If the base is a List, the propertyResolved property of the ELContext object must be set to true by this resolver, before returning. If this property is not true after this method is called, the caller can safely assume no value was set.

If this resolver was constructed in read-only mode, this method will always throw PropertyNotWritableException.

If a List was created using java.util.Collections.unmodifiableList(List), this method must throw PropertyNotWritableException. Unfortunately, there is no Collections API method to detect this. However, an implementation can create a prototype unmodifiable List and query its runtime type to see if it matches the runtime type of the base object as a workaround.

Overrides: setValue in class ELResolver

Parameters:
- context - The context of this evaluation.
- base - The list to be modified. Only bases of type List are handled by this resolver.
- property - The index of the value to be set. Will be coerced into an integer.
- val - The value to be set at the given index.

Throws:
- java.lang.ClassCastException - if the class of the specified element prevents it from being added to this list.
- java.lang.NullPointerException - if context is null, or if the value is null and this List does not support null elements.
- java.lang.IllegalArgumentException - if the property could not be coerced into an integer, or if some aspect of the specified element prevents it from being added to this list.
- PropertyNotWritableException - if this resolver was constructed in read-only mode, or if the set operation is not supported by the underlying list.
- ELException - if an exception was thrown while performing the property or variable resolution. The thrown exception must be included as the cause property of this exception, if available.
2.15 javax.el
MapELResolver

2.15.1 Declaration

public class MapELResolver extends ELResolver

java.lang.Object
  |---javax.el.ELResolver
  |   |---javax.el.MapELResolver

2.15.2 Description

Defines property resolution behavior on instances of java.util.Map.

This resolver handles base objects of type java.util.Map. It accepts any object as a property and uses that object as a key in the map. The resulting value is the value in the map that is associated with that key.

This resolver can be constructed in read-only mode, which means that isReadOnly will always return true and setValue(ELContext, Object, Object, Object) will always throw PropertyNotWritableException.

ELResolvers are combined together using CompositeELResolver, to define rich semantics for evaluating an expression. See the javadocs for ELResolver for details.

Since: JSP 2.1

See Also: CompositeELResolver, ELResolver, java.util.Map

### Member Summary

<table>
<thead>
<tr>
<th>Constructors</th>
<th></th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>MapELResolver()</td>
<td>getCommonPropertyType(ELContext context, java.lang.Object base)</td>
<td></td>
</tr>
<tr>
<td>MapELResolver(boolean isReadOnly)</td>
<td>getFeatureDescriptors(ELContext context, java.lang.Object base)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getFeatureDescriptors(ELContext context, java.lang.Object base)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getType(ELContext context, java.lang.Object base, java.lang.Object property)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getValue(ELContext context, java.lang.Object base, java.lang.Object property)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>isReadOnly(ELContext context, java.lang.Object base, java.lang.Object property)</td>
<td></td>
</tr>
</tbody>
</table>
Inherited Member Summary

Fields inherited from class ELResolver
RESOLVABLE_AT_DESIGN_TIME, TYPE

Methods inherited from class Object
clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)

Constructors

2.15.3 MapELResolver()

public MapELResolver()

Creates a new read/write MapELResolver.

2.15.4 MapELResolver(boolean)

public MapELResolver(boolean isReadOnly)

Creates a new MapELResolver whose read-only status is determined by the given parameter.

Parameters:

isReadOnly - true if this resolver cannot modify maps; false otherwise.

Methods

2.15.5 getCommonPropertyType(ELContext, Object)

public java.lang.Class<?> getCommonPropertyType(javax.el.ELContext context, java.lang.Object base)

If the base object is a map, returns the most general type that this resolver accepts for the property argument. Otherwise, returns null.

Assuming the base is a Map, this method will always return Object.class. This is because Maps accept any object as a key.

Overrides: getCommonPropertyType in class ELResolver

Parameters:

context - The context of this evaluation.

base - The map to analyze. Only bases of type Map are handled by this resolver.

Returns: null if base is not a Map; otherwise Object.class.

2.15.6 getFeatureDescriptors(ELContext, Object)

public java.util.Iterator<FeatureDescriptor> getFeatureDescriptors(javax.el.ELContext context, java.lang.Object base)

If the base object is a map, returns an Iterator containing the set of keys available in the Map. Otherwise, returns null.
The `Iterator` returned must contain zero or more instances of `java.beans.FeatureDescriptor`. Each info object contains information about a key in the Map, and is initialized as follows:

- `displayName` - The return value of calling the `toString` method on this key, or "null" if the key is `null`.
- `name` - Same as `displayName` property.
- `shortDescription` - Empty string
- `expert` - false
- `hidden` - false
- `preferred` - true

In addition, the following named attributes must be set in the returned `FeatureDescriptor`s:

- `ELResolver.TYPE` - The return value of calling the `getClass()` method on this key, or `null` if the key is `null`.
- `ELResolver.RESOLVABLE_AT_DESIGN_TIME` - true

**Overrides:** `getFeatureDescriptors` in class `ELResolver`

**Parameters:**
- `context` - The context of this evaluation.
- `base` - The map whose keys are to be iterated over. Only bases of type `Map` are handled by this resolver.

**Returns:** An `Iterator` containing zero or more (possibly infinitely more) `FeatureDescriptor` objects, each representing a key in this map, or `null` if the base object is not a map.

### 2.15.7 `getType(ELContext, Object, Object)`

```java
public java.lang.Class<?> getType(javax.el.ELContext context, java.lang.Object base, java.lang.Object property)
```

If the base object is a map, returns the most general acceptable type for a value in this map.

If the base is a `Map`, the `propertyResolved` property of the `ELContext` object must be set to `true` by this resolver, before returning. If this property is not `true` after this method is called, the caller should ignore the return value.

Assuming the base is a `Map`, this method will always return `Object.class`. This is because Maps accept any object as the value for a given key.

**Overrides:** `getType` in class `ELResolver`

**Parameters:**
- `context` - The context of this evaluation.
- `base` - The map to analyze. Only bases of type `Map` are handled by this resolver.
- `property` - The key to return the acceptable type for. Ignored by this resolver.

**Returns:** If the `propertyResolved` property of `ELContext` was set to `true`, then the most general acceptable type; otherwise undefined.

**Throws:**
- `java.lang.NullPointerException` - if `context` is `null`
- `ELException` - if an exception was thrown while performing the property or variable resolution.

The thrown exception must be included as the cause property of this exception, if available.

### 2.15.8 `getValue(ELContext, Object, Object)`

```java
```


If the base object is a map, returns the value associated with the given key, as specified by the property argument. If the key was not found, null is returned.

If the base is a Map, the propertyResolved property of the ELContext object must be set to true by this resolver, before returning. If this property is not true after this method is called, the caller should ignore the return value.

Just as in java.util.Map.get(Object), just because null is returned doesn’t mean there is no mapping for the key; it’s also possible that the Map explicitly maps the key to null.

Overrides: getValue in class ELResolver

Parameters:
  context - The context of this evaluation.
  base - The map to be analyzed. Only bases of type Map are handled by this resolver.
  property - The key whose associated value is to be returned.

Returns: If the propertyResolved property of ELContext was set to true, then the value associated with the given key or null if the key was not found. Otherwise, undefined.

Throws:
  java.lang.ClassCastException - if the key is of an inappropriate type for this map (optionally thrown by the underlying Map).
  java.lang.NullPointerException - if context is null, or if the key is null and this map does not permit null keys (the latter is optionally thrown by the underlying Map).
  ELException - if an exception was thrown while performing the property or variable resolution. The thrown exception must be included as the cause property of this exception, if available.

2.15.9 isReadOnly(ELContext, Object, Object)

public boolean isReadOnly(javax.el.ELContext context, java.lang.Object base, java.lang.Object property)

If the base object is a map, returns whether a call to setValue(ELContext, Object, Object, Object) will always fail.

If the base is a Map, the propertyResolved property of the ELContext object must be set to true by this resolver, before returning. If this property is not true after this method is called, the caller should ignore the return value.

If this resolver was constructed in read-only mode, this method will always return true.

If a Map was created using java.util.Collections.unmodifiableMap(Map), this method must return true. Unfortunately, there is no Collections API method to detect this. However, an implementation can create a prototype unmodifiable Map and query its runtime type to see if it matches the runtime type of the base object as a workaround.

Overrides: isReadOnly in class ELResolver

Parameters:
  context - The context of this evaluation.
  base - The map to analyze. Only bases of type Map are handled by this resolver.
  property - The key to return the read-only status for. Ignored by this resolver.
javax.el MapELResolver

setValue(ELContext, Object, Object, Object)

Returns: If the propertyResolved property of ELContext was set to true, then true if calling the setValue method will always fail or false if it is possible that such a call may succeed; otherwise undefined.

Throws:
- java.lang.NullPointerException - if context is null
- ELException - if an exception was thrown while performing the property or variable resolution. The thrown exception must be included as the cause property of this exception, if available.

2.15.10 setValue(ELContext, Object, Object, Object)

public void setValue(ELContext context, Object base, Object property, Object val)

If the base object is a map, attempts to set the value associated with the given key, as specified by the property argument.

If the base is a Map, the propertyResolved property of the ELContext object must be set to true by this resolver, before returning. If this property is not true after this method is called, the caller can safely assume no value was set.

If this resolver was constructed in read-only mode, this method will always throw PropertyNotWritableException.

If a Map was created using java.util.Collections.unmodifiableMap(Map), this method must throw PropertyNotWritableException. Unfortunately, there is no Collections API method to detect this. However, an implementation can create a prototype unmodifiable Map and query its runtime type to see if it matches the runtime type of the base object as a workaround.

Overrides: setValue in class ELResolver

Parameters:
- context - The context of this evaluation.
- base - The map to be modified. Only bases of type Map are handled by this resolver.
- property - The key with which the specified value is to be associated.
- val - The value to be associated with the specified key.

Throws:
- java.lang.ClassCastException - if the class of the specified key or value prevents it from being stored in this map.
- java.lang.NullPointerException - if context is null, or if this map does not permit null keys or values, and the specified key or value is null.
- java.lang.IllegalArgumentException - if some aspect of this key or value prevents it from being stored in this map.
- ELException - if an exception was thrown while performing the property or variable resolution. The thrown exception must be included as the cause property of this exception, if available.
- PropertyNotWritableException - if this resolver was constructed in read-only mode, or if the put operation is not supported by the underlying map.
MethodExpression javax.el

setValue(ELContext, Object, Object, Object)

2.16 javax.el
MethodExpression

2.16.1 Declaration
public abstract class MethodExpression extends Expression

java.lang.Object
  |---javax.el.Expression
  |       |---javax.el.MethodExpression

All Implemented Interfaces: java.io.Serializable

2.16.2 Description
An Expression that refers to a method on an object.

The ExpressionFactory.createMethodExpression(ELContext, String, Class, Class[]) method can be used to parse an expression string and return a concrete instance of MethodExpression that encapsulates the parsed expression. The FunctionMapper is used at parse time, not evaluation time, so one is not needed to evaluate an expression using this class. However, the ELContext is needed at evaluation time.

The getMethodInfo(ELContext) and invoke(ELContext, Object[]) methods will evaluate the expression each time they are called. The ELResolver in the ELContext is used to resolve the top-level variables and to determine the behavior of the . and [] operators. For any of the two methods, the ELResolver.getValue(ELContext, Object, Object) method is used to resolve all properties up to but excluding the last one. This provides the base object on which the method appears. If the base object is null, a PropertyNotFoundException must be thrown. At the last resolution, the final property is then coerced to a String, which provides the name of the method to be found. A method matching the name and expected parameters provided at parse time is found and it is either queried or invoked (depending on the method called on this MethodExpression).

See the notes about comparison, serialization and immutability in the Expression javadocs.

Since: JSP 2.1

See Also: ELResolver, Expression, ExpressionFactory

Member Summary

| Constructors | MethodExpression() |
| Methods      | getMethodInfo(ELContext context) |
|              | invoke(ELContext context, java.lang.Object[] params) |
Inherited Member Summary

Methods inherited from class Expression
equals(Object), getExpressionString(), hashCode(), isLiteralText()

Methods inherited from class Object
close(), finalize(), getClass(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)

Constructors

2.16.3 MethodExpression()

public MethodExpression()

Methods

2.16.4 getMethodInfo(ELContext)

public abstract javax.el.MethodInfo getMethodInfo(javax.el.ELContext context)

Evaluates the expression relative to the provided context, and returns information about the actual referenced method.

Parameters:
context - The context of this evaluation

Returns: an instance of MethodInfo containing information about the method the expression evaluated to.

Throws:
java.lang.NullPointerException - if context is null
PropertyNotFoundException - if one of the property resolutions failed because a specified variable or property does not exist or is not readable.
MethodNotFoundException - if no suitable method can be found.
ELException - if an exception was thrown while performing property or variable resolution. The thrown exception must be included as the cause property of this exception, if available.

2.16.5 invoke(ELContext, Object[])

public abstract java.lang.Object invoke(javax.el.ELContext context, java.lang.Object[] params)

If a String literal is specified as the expression, returns the String literal coerced to the expected return type of the method signature. An ELException is thrown if expectedReturnType is void or if the coercion of the String literal to the expectedReturnType yields an error (see Section “1.16 Type Conversion” of the EL specification). If not a String literal, evaluates the expression relative to the provided
MethodExpression
javax.el
invoke(ELContext, Object[])

context, invokes the method that was found using the supplied parameters, and returns the result of the
method invocation. Any parameters passed to this method is ignored if isLiteralText() is true.

Parameters:
   context - The context of this evaluation.
   params - The parameters to pass to the method, or null if no parameters.

Returns: the result of the method invocation (null if the method has a void return type).

Throws:
   java.lang.NullPointerException - if context is null
   PropertyNotFoundException - if one of the property resolutions failed because a specified
variable or property does not exist or is not readable.
   MethodNotFoundException - if no suitable method can be found.
   ELException - if a String literal is specified and expectedReturnType of the MethodExpression is
void or if the coercion of the String literal to the expectedReturnType yields an error (see Section “1.16
Type Conversion”).
   ELException - if an exception was thrown while performing property or variable resolution. The
thrown exception must be included as the cause property of this exception, if available. If the exception
thrown is an InvocationTargetException, extract its cause and pass it to the
ELException constructor.
2.17 javax.el

MethodInfo

2.17.1 Declaration

public class MethodInfo

java.lang.Object
    | +--javax.el.MethodInfo

2.17.2 Description

Holds information about a method that a MethodExpression evaluated to.

Since: JSP 2.1

Member Summary

Constructors

MethodInfo(java.lang.String name, java.lang.Class<?> returnType, java.lang.Class<?>[] paramTypes)

Methods

java.lang.String getName()
java.lang.Class<?>[] getParamTypes()
java.lang.Class<?> getReturnType()

Inherited Member Summary

Methods inherited from class Object

clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)

Constructors

2.17.3 MethodInfo(String, Class<?>, Class<?>[][])

public MethodInfo(java.lang.String name, java.lang.Class<?> returnType, java.lang.Class<?>[][] paramTypes)

Creates a new instance of MethodInfo with the given information.

Parameters:

name - The name of the method
returnType - The return type of the method
Methods

2.17.4 getName()

    public java.lang.String getName()

Returns the name of the method

Returns: the name of the method

2.17.5 getParamTypes()

    public java.lang.Class<?>[] getParamTypes()

Returns the parameter types of the method

Returns: the parameter types of the method

2.17.6 getReturnType()

    public java.lang.Class<? extends java.lang.Object> getReturnType()

Returns the return type of the method

Returns: the return type of the method
2.18  javax.el

MethodNotFoundException

2.18.1 Declaration

public class MethodNotFoundException extends ELException

java.lang.Object
    |---java.lang.Throwable
        |---java.lang.Exception
            |---java.lang.RuntimeException
                |---javax.el.ELException
                    |---javax.el.MethodNotFoundException

All Implemented Interfaces: java.io.Serializable

2.18.2 Description

Thrown when a method could not be found while evaluating a MethodExpression.

Since: JSP 2.1

See Also: MethodExpression

<table>
<thead>
<tr>
<th>Constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td>MethodNotFoundException()</td>
</tr>
<tr>
<td>MethodNotFoundException(java.lang.String message)</td>
</tr>
<tr>
<td>MethodNotFoundException(java.lang.String pMessage, java.lang.Throwable pRootCause)</td>
</tr>
<tr>
<td>MethodNotFoundException(java.lang.Throwable exception)</td>
</tr>
</tbody>
</table>

Inherited Member Summary

Methods inherited from class Object

clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), wait(), wait(long), wait(long, int)

Methods inherited from class Throwable

fillInStackTrace(), getCause(), getLocalizedMessage(), getMessage(), getStackTrace(), initCause(Throwable), printStackTrace(), printStackTrace(PrintStream), printStackTrace(PrintWriter), setStackTrace(StackTraceElement[]), toString()
Constructors

2.18.3 MethodNotFoundException()

public MethodNotFoundException()

Creates a MethodNotFoundException with no detail message.

2.18.4 MethodNotFoundException(String)

public MethodNotFoundException(java.lang.String message)

Creates a MethodNotFoundException with the provided detail message.

Parameters:
message - the detail message

2.18.5 MethodNotFoundException(Throwable)

public MethodNotFoundException(java.lang.Throwable exception)

Creates a MethodNotFoundException with the given root cause.

Parameters:
exception - the originating cause of this exception

2.18.6 MethodNotFoundException(String, Throwable)

public MethodNotFoundException(java.lang.String pMessage, java.lang.Throwable pRootCause)

Creates a MethodNotFoundException with the given detail message and root cause.

Parameters:
pMessage - the detail message
pRootCause - the originating cause of this exception
2.19javax.el

PropertyNotFoundException

2.19.1Declaration
public class PropertyNotFoundException extends ELException

java.lang.Object
   +--java.lang.Throwable
      +--java.lang.Exception
         +--java.lang.RuntimeException
            +--javax.el.ELException
               +--javax.el.PropertyNotFoundException

All Implemented Interfaces: java.io.Serializable

2.19.2Description
Thrown when a property could not be found while evaluating a ValueExpression or MethodExpression.

For example, this could be triggered by an index out of bounds while setting an array value, or by an unreadable property while getting the value of a JavaBeans property.

Since: JSP 2.1

<table>
<thead>
<tr>
<th>Member Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructors</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>PropertyNotFoundException() 70</td>
</tr>
<tr>
<td>PropertyNotFoundException(java.lang.String message) 70</td>
</tr>
<tr>
<td>PropertyNotFoundException(java.lang.String pMessage, java.lang.Throwable pRootCause) 70</td>
</tr>
<tr>
<td>PropertyNotFoundException(java.lang.Throwable exception) 70</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inherited Member Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from class Object</td>
</tr>
<tr>
<td>clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), wait(), wait(long), wait(long, int)</td>
</tr>
<tr>
<td>Methods inherited from class Throwable</td>
</tr>
</tbody>
</table>

javax.el PropertyNotFoundException
MethodNotFoundException(String, Throwable)
2.19.3 PropertyNotFoundException()

public PropertyNotFoundException()

Creates a PropertyNotFoundException with no detail message.

2.19.4 PropertyNotFoundException(String)

public PropertyNotFoundException(java.lang.String message)

Creates a PropertyNotFoundException with the provided detail message.

Parameters:
message - the detail message

2.19.5 PropertyNotFoundException(Throwable)

public PropertyNotFoundException(java.lang.Throwable exception)

Creates a PropertyNotFoundException with the given root cause.

Parameters:
exception - the originating cause of this exception

2.19.6 PropertyNotFoundException(String, Throwable)

public PropertyNotFoundException(java.lang.String pMessage,
java.lang.Throwable pRootCause)

Creates a PropertyNotFoundException with the given detail message and root cause.

Parameters:
pMessage - the detail message
pRootCause - the originating cause of this exception
2.20  javax.el

PropertyNotWritableException

2.20.1 Declaration

public class PropertyNotWritableException extends ELException

java.lang.Object
  |  --java.lang.Throwable
  |   |  --java.lang.Exception
  |   |   |  --java.lang.RuntimeException
  |   |   |   |  --javax.el.ELException
  |   |   |   |   |  --javax.el.PropertyNotWritableException

All Implemented Interfaces: java.io.Serializable

2.20.2 Description

Thrown when a property could not be written to while setting the value on a ValueExpression.
For example, this could be triggered by trying to set a map value on an unmodifiable map.

Since:  JSP 2.1

<table>
<thead>
<tr>
<th>Constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>PropertyNotWritableException()</td>
</tr>
<tr>
<td>PropertyNotWritableException(java.lang.String pMessage)</td>
</tr>
<tr>
<td>PropertyNotWritableException(java.lang.String pMessage, java.lang.Throwable pRootCause)</td>
</tr>
<tr>
<td>PropertyNotWritableException(java.lang.Throwable exception)</td>
</tr>
</tbody>
</table>

Inherited Member Summary

Methods inherited from class Object

clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), 
wait(), wait(long), wait(long, int)

Methods inherited from class Throwable

fillInStackTrace(), getCause(), getLocalizedMessage(), getMessage(), getStackTrace(), 
initCause(Throwable), printStackTrace(), printStackTrace(PrintStream), 
printStackTrace(PrintWriter), setStackTrace(StackTraceElement[]), toString()
Constructors

2.20.3 PropertyNotWritableException()

public PropertyNotWritableException()

Creates a PropertyNotWritableException with no detail message.

2.20.4 PropertyNotWritableException(String)

public PropertyNotWritableException(java.lang.String pMessage)

Creates a PropertyNotWritableException with the provided detail message.

Parameters:
   pMessage - the detail message

2.20.5 PropertyNotWritableException(Throwable)

public PropertyNotWritableException(java.lang.ThrowException) exception

Creates a PropertyNotWritableException with the given root cause.

Parameters:
   exception - the originating cause of this exception

2.20.6 PropertyNotWritableException(String, Throwable)

public PropertyNotWritableException(java.lang.String pMessage, java.lang.ThrowException pRootCause)

Creates a PropertyNotWritableException with the given detail message and root cause.

Parameters:
   pMessage - the detail message
   pRootCause - the originating cause of this exception
2.21 javax.el

ResourceBundleELResolver

2.21.1 Declaration

```java
class ResourceBundleELResolver extends ELResolver
```

2.21.2 Description

Defines property resolution behavior on instances of `java.util.ResourceBundle`. This resolver handles base objects of type `java.util.ResourceBundle`. It accepts any object as a property and coerces it to a `java.lang.String` for invoking `java.util.ResourceBundle.getObject(String)`. This resolver is read only and will throw a `PropertyNotWritableException` if `setValue` is called.

ELResolvers are combined together using `CompositeELResolver`, to define rich semantics for evaluating an expression. See the javadocs for `ELResolver` for details.

**Since:** JSP 2.1

**See Also:** `CompositeELResolver`, `ELResolver`, `java.util.ResourceBundle`

---

### Member Summary

<table>
<thead>
<tr>
<th>Constructors</th>
<th>ResourceBundleELResolver()</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>java.lang.Class&lt;?&gt;</code></td>
<td>getCommonPropertyType(ELContext context, java.lang.Object base)</td>
</tr>
<tr>
<td><code>java.util.Iterator&lt;FeatureDescriptor&gt;</code></td>
<td>getFeatureDescriptors(ELContext context, java.lang.Object base)</td>
</tr>
<tr>
<td><code>java.lang.Class&lt;?&gt;</code></td>
<td>getType(ELContext context, java.lang.Object base, java.lang.Object property)</td>
</tr>
<tr>
<td><code>java.lang.Object</code></td>
<td>getValue(ELContext context, java.lang.Object base, java.lang.Object property)</td>
</tr>
<tr>
<td><code>boolean</code></td>
<td>isReadOnly(ELContext context, java.lang.Object base, java.lang.Object property)</td>
</tr>
</tbody>
</table>
Constructions

2.21.3 ResourceBundleELResolver()

    public ResourceBundleELResolver()

Methods

2.21.4 getCommonPropertyType(ELContext, Object)

    public java.lang.Class<?> getCommonPropertyType(javax.el.ELContext context, java.lang.Object base)

    If the base object is a ResourceBundle, returns the most general type that this resolver accepts for the property argument. Otherwise, returns null.

Assuming the base is a ResourceBundle, this method will always return String.class.

Overrides: getCommonPropertyType in class ELResolver

Parameters:
    context - The context of this evaluation.
    base - The bundle to analyze. Only bases of type ResourceBundle are handled by this resolver.

Returns: null if base is not a ResourceBundle; otherwise String.class.

2.21.5 getFeatureDescriptors(ELContext, Object)

    public java.util.Iterator<java.beans.FeatureDescriptor> getFeatureDescriptors(javax.el.ELContext context, java.lang.Object base)

    If the base object is a ResourceBundle, returns an Iterator containing the set of keys available in the ResourceBundle. Otherwise, returns null.

The Iterator returned must contain zero or more instances of java.beans.FeatureDescriptor. Each info object contains information about a key in the ResourceBundle, and is initialized as follows:

displayName - The String key name - Same as displayName property. shortDescription - Empty string expert - false hidden - false preferred - true

In addition, the following named attributes must be set in the returned FeatureDescriptors:

ELResolver.TYPE - String.class ELResolver.RESOLVABLE_AT_DESIGN_TIME - true
Overrides: `getFeatureDescriptors()` in class `ELResolver`  

Parameters:  
- `context` - The context of this evaluation.  
- `base` - The bundle whose keys are to be iterated over. Only bases of type `ResourceBundle` are handled by this resolver.  

Returns: An `Iterator` containing zero or more (possibly infinitely more) `FeatureDescriptor` objects, each representing a key in this bundle, or `null` if the base object is not a `ResourceBundle`.  

2.21.6 `getType(ELContext, Object, Object)`  

```java
public java.lang.Class<?> getType(javax.el.ELContext context, java.lang.Object base, java.lang.Object property)
```

If the base object is an instance of `ResourceBundle`, return `null`, since the resolver is read only.

If the base is `ResourceBundle`, the `propertyResolved` property of the `ELContext` object must be set to `true` by this resolver, before returning. If this property is not `true` after this method is called, the caller should ignore the return value.

Overrides: `getType()` in class `ELResolver`  

Parameters:  
- `context` - The context of this evaluation.  
- `base` - The `ResourceBundle` to analyze.  
- `property` - The name of the property to analyze.  

Returns: If the `propertyResolved` property of `ELContext` was set to `true`, then `null`; otherwise undefined.  

Throws:  
- `java.lang.NullPointerException` - if `context` is `null`  

2.21.7 `getValue(ELContext, Object, Object)`  

```java
```

If the base object is an instance of `ResourceBundle`, the provided property will first be coerced to a `String`. The `Object` returned by `getObject` on the base `ResourceBundle` will be returned.

If the base is `ResourceBundle`, the `propertyResolved` property of `ELContext` object must be set to `true` by this resolver, before returning. If this property is not `true` after this method is called, the caller should ignore the return value.

Overrides: `getValue()` in class `ELResolver`  

Parameters:  
- `context` - The context of this evaluation.  
- `base` - The `ResourceBundle` to analyze.  
- `property` - The name of the property to analyze. Will be coerced to a `String`.  

Returns: If the `propertyResolved` property of `ELContext` was set to `true`, then `null` if property is `null`; otherwise the `Object` for the given key (property coerced to `String`) from the `ResourceBundle`. If no object for the given key can be found, then the `String` “???” + key + “???”.
2.21.8 isReadOnly(ELContext, Object, Object)

public boolean isReadOnly(javax.el.ELContext context, java.lang.Object base, java.lang.Object property)

If the base object is not null and an instanceof java.util.ResourceBundle, return true.

Overrides: isReadOnly in class ELResolver

Parameters:
- context - The context of this evaluation.
- base - The ResourceBundle to be modified. Only bases that are of type ResourceBundle are handled.
- property - The String property to use.

Returns: If the propertyResolved property of ELContext was set to true, then true; otherwise undefined.

Throws:
- java.lang.NullPointerException - if context is null

2.21.9 setValue(ELContext, Object, Object, Object)


If the base object is a ResourceBundle, throw a PropertyNotWritableException.

Overrides: setValue in class ELResolver

Parameters:
- context - The context of this evaluation.
- base - The ResourceBundle to be modified. Only bases that are of type ResourceBundle are handled.
- property - The String property to use.
- value - The value to be set.

Throws:
- java.lang.NullPointerException - if context is null.
- PropertyNotWritableException - Always thrown if base is an instance of ReasourceBundle.
2.22 javax.el

ValueExpression

2.22.1 Declaration

public abstract class ValueExpression extends Expression

java.lang.Object
  +--javax.el.Expression
    +--javax.el.ValueExpression

All Implemented Interfaces: java.io.Serializable

2.22.2 Description

An Expression that can get or set a value.

In previous incarnations of this API, expressions could only be read. ValueExpression objects can now be used both to retrieve a value and to set a value. Expressions that can have a value set on them are referred to as l-value expressions. Those that cannot are referred to as r-value expressions. Not all r-value expressions can be used as l-value expressions (e.g. "${1+1}" or "${firstName} ${lastName}"). See the EL Specification for details. Expressions that cannot be used as l-values must always return true from isReadOnly().

The ExpressionFactory.createValueExpression(ELContext, String, Class) method can be used to parse an expression string and return a concrete instance of ValueExpression that encapsulates the parsed expression. The FunctionMapper is used at parse time, not evaluation time, so one is not needed to evaluate an expression using this class. However, the ELContext is needed at evaluation time.

The getValue(ELContext), setValue(ELContext, Object), isReadOnly(ELContext) methods will evaluate the expression each time they are called. The ELResolver in the ELContext is used to resolve the top-level variables and to determine the behavior of the . and [] operators. For any of the four methods, the ELResolver.getValue(ELContext, Object, Object), ELResolver.setValue(ELContext, Object, Object, Object), ELResolver.isReadOnly(ELContext, Object, Object) or ELResolver.getType(ELContext, Object, Object) method, depending on which was called on the ValueExpression.

See the notes about comparison, serialization and immutability in the Expression javadocs.

Since: JSP 2.1

See Also: ELResolver, Expression, ExpressionFactory
## Member Summary

### Constructors

#### 2.22.3 ValueExpression()

```java
public ValueExpression()
```

### Methods

#### 2.22.4 getExpectedType()

```java
public abstract java.lang.Class<?> getExpectedType()
```

Returns the type the result of the expression will be coerced to after evaluation.

**Returns:** the `expectedType` passed to the `ExpressionFactory.createValueExpression` method that created this `ValueExpression`.

#### 2.22.5 getType(ELContext)

```java
public abstract java.lang.Class<?> getType(javax.el.ELContext context)
```

Evaluates the expression relative to the provided context, and returns the most general type that is acceptable for an object to be passed as the `value` parameter in a future call to the `setValue(ELContext, Object)` method.

This is not always the same as `getValue().getClass()`. For example, in the case of an expression that references an array element, the `getType` method will return the element type of the array, which might be a superclass of the type of the actual element that is currently in the specified array element.

---

**Inherited Member Summary**

### Methods inherited from class `Expression`

- `equals(Object)`
- `getExpressionString()`
- `hashCode()`
- `isLiteralText()`

### Methods inherited from class `Object`

- `clone()`, `finalize()`, `getClass()`, `notify()`, `notifyAll()`, `toString()`, `wait()`, `wait(long)`, `wait(long, int)`
Parameters:
  context - The context of this evaluation.

Returns: the most general acceptable type; otherwise undefined.

Throws:
  java.lang.NullPointerException - if context is null.
  PropertyNotFoundException - if one of the property resolutions failed because a specified variable or property does not exist or is not readable.
  ELException - if an exception was thrown while performing property or variable resolution. The thrown exception must be included as the cause property of this exception, if available.

2.22.6 getValue(ELContext)

public abstract java.lang.Object getValue(javax.el.ELContext context)

Evaluates the expression relative to the provided context, and returns the resulting value.

The resulting value is automatically coerced to the type returned by getExpectedType(), which was provided to the ExpressionFactory when this expression was created.

Parameters:
  context - The context of this evaluation.

Returns: The result of the expression evaluation.

Throws:
  java.lang.NullPointerException - if context is null.
  PropertyNotFoundException - if one of the property resolutions failed because a specified variable or property does not exist or is not readable.
  ELException - if an exception was thrown while performing property or variable resolution. The thrown exception must be included as the cause property of this exception, if available.

2.22.7 isReadOnly(ELContext)

public abstract boolean isReadOnly(javax.el.ELContext context)

Evaluates the expression relative to the provided context, and returns true if a call to setValue(ELContext, Object) will always fail.

Parameters:
  context - The context of this evaluation.

Returns: true if the expression is read-only or false if not.

Throws:
  java.lang.NullPointerException - if context is null.
  PropertyNotFoundException - if one of the property resolutions failed because a specified variable or property does not exist or is not readable.
  ELException - if an exception was thrown while performing property or variable resolution. The thrown exception must be included as the cause property of this exception, if available. * @throws NullPointerException if context is null

2.22.8 setValue(ELContext, Object)

public abstract void setValue(javax.el.ELContext context, java.lang.Object value)
ValueExpression javax.el
setValue(ELContext, Object)

Evaluates the expression relative to the provided context, and sets the result to the provided value.

Parameters:

class context - The context of this evaluation.

class value - The new value to be set.

Throws:

java.lang.NullPointerException - if context is null.

PropertyNotFoundException69 - if one of the property resolutions failed because a specified variable or property does not exist or is not readable.

PropertyNotWritableException71 - if the final variable or property resolution failed because the specified variable or property is not writable.

ELException35 - if an exception was thrown while attempting to set the property or variable. The thrown exception must be included as the cause property of this exception, if available.
2.23 javax.el

VariableMapper

2.23.1 Declaration
public abstract class VariableMapper

java.lang.Object
   +--javax.el.VariableMapper

2.23.2 Description
The interface to a map between EL variables and the EL expressions they are associated with.

Since: JSP 2.1

Member Summary

<table>
<thead>
<tr>
<th>Constructors</th>
<th>VariableMapper()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods</td>
<td></td>
</tr>
<tr>
<td>abstract resolveVariable(java.lang.String variable)</td>
<td></td>
</tr>
<tr>
<td>ValueExpression</td>
<td></td>
</tr>
<tr>
<td>abstract setVariable(java.lang.String variable, ValueExpression expression)</td>
<td></td>
</tr>
<tr>
<td>ValueExpression</td>
<td></td>
</tr>
</tbody>
</table>

Inherited Member Summary

Methods inherited from class Object
clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)

Constructors

2.23.3 VariableMapper()

public VariableMapper()

Methods

2.23.4 resolveVariable(String)

public abstract javax.el.ValueExpression resolveVariable(java.lang.String variable)
Parameters:
    variable - The variable name

Returns: the ValueExpression assigned to the variable, null if there is no previous assignment to this variable.

2.23.5 setVariable(String, ValueExpression)

    public abstract javax.el.ValueExpression setVariable(java.lang.String variable,
              javax.el.ValueExpression expression)

Assign a ValueExpression to an EL variable, replacing any previously assignment to the same variable. The assignment for the variable is removed if the expression is null.

Parameters:
    variable - The variable name
    expression - The ValueExpression to be assigned to the variable.

Returns: The previous ValueExpression assigned to this variable, null if there is no previous assignment to this variable.
CHAPTER A

Changes

This appendix lists the changes in the EL specification. This appendix is non-normative.

A.1 Changes between Final Release and Proposed Final Draft 2

Added support for enumerated data types. Coercions and comparisons were updated to include enumerated type types.

A.2 Changes between Public Review and Proposed Final Draft

New constructor for derived exception classes

Exception classes that extend `ELException` (PropertyNotFoundException, PropertyNotWritableException, MethodNotFoundException) did not have a constructor with both 'message' and 'rootCause' as arguments (as it exists in `ELException`). The constructor has been added to these classes.

**javax.el.ELContext API changes**

- removed the `ELContext` constructor
  ```java
  protected ELContext(javax.el.ELResolver resolver)
  ```
- added the following abstract method in `ELContext`
  ```java
  public abstract javax.el.ELResolver getELResolver();
  ```
Section 1.8.1 - A {<,>,<,>,=,lt,gt,le,ge} B

- If the first condition (A==B) is false, simply fall through to the next step (do not return false). See issue 129 at jsp-spec-public.dev.java.net.

javax.el.ResourceBundleELResolver

- New ELResolver class added to support easy access to localized messages.

Generics

- Since JSP 2.1 requires J2SE 5.0, we’ve modified the APIs that can take advantage of generics. These include:
  - `ExpressionFactory:createValueExpression()`,
  - `ExpressionFactory:createMethodExpression()`,
  - `ExpressionFactory:coerceToType()`, `ELResolver:getType()`,
  - `ELResolver:getCommonPropertyType()`, `MethodInfo:MethodInfo()`,
  - `MethodInfo:returnType()`, `MethodInfo:getParamTypes()`

A.3 Changes between Early Draft Release and Public Review

New concept: EL Variables

The EL now supports the concept of EL Variables to properly support code structures such as `<c:forEach>` where a nested action accesses a deferred expression that includes a reference to an iteration variable.

- Resulting API changes are:
  - The `javax.el` package description describes the motivation behind EL variables.
  - `ElContext` has two additional methods to provide access to `FunctionMapper` and `VariableMapper`.
  - `ExpressionFactory` creation methods now take an `ELContext` parameter. `FunctionMapper` has been removed as a parameter to these methods.
  - Added new class `VariableMapper`

- At a few locations in the spec, the term “variable” has been replaced with “model object” to avoid confusion between model objects and the newly introduced EL variables.

- Added new section “Variables” after section 1.15 to introduce the concept of EL Variables.
EL in a nutshell (section 1.1.1)
- Added a paragraph commenting on the flexibility of the EL, thanks to its pluggable API for the resolution of model objects, functions, and variables.

javax.el.EJException
- ElException now extends RuntimeException instead of Exception.
- Method getRootCause() has been removed in favor of Throwable.getCause().

javax.el.ExpressionFactory
- Creation methods now use ELContext instead of FunctionMapper (see EL Variables above).
- Added method coerceToType(). See issue 132 at jsp-spec-public.dev.java.net.

javax.el.MethodExpression
- invoke() must unwrap an InvocationTargetException before re-throwing as an ELException.

Section 1.6 - Operators [] and .
- PropertyNotFoundException is now thrown instead of NullPointerException when this is the last property being resolved and we’re dealing with an lvalue that is null.

Section 1.13 - Operator Precedence
- Clarified the fact that qualified functions with a namespace prefix have precedence over the operators.

Faces Action Attribute and MethodExpression
In Faces, the action attribute accepts both a String literal or a MethodExpression. When migrating to JSF 1.2, if the attribute's type is set as MethodExpression, an error would be reported if a String literal is specified because a String literal cannot evaluate to a valid javax.el.MethodExpression.

To solve this issue, the specification of MethodExpression has been expanded to also support String literal-expressions. Changes have been made to:
- Section 1.2.2
- ExpressionFactory.createMethodExpression()
- javax.el.MethodExpression:invoke()