Expression Language

This chapter introduces the Expression Language (also referred to as the EL), which provides an important mechanism for enabling the presentation layer (web pages) to communicate with the application logic (backing beans).

The EL is used by both JavaServer Faces technology and JavaServer Pages (JSP) technology.

The EL represents a union of the expression languages offered by JavaServer Faces technology and JSP technology.

The following topics are addressed here:

- Overview of the EL
- Immediate and Deferred Evaluation Syntax
- Value and Method Expressions
- Defining a Tag Attribute Type
- Literal Expressions
- Operators
- Reserved Words
- Examples of EL Expressions

Overview of the EL

The EL allows page authors to use simple expressions to dynamically access data from JavaBeans components.

For example, the `test` attribute of the following conditional tag is supplied with an EL expression that compares 0 with the number of items in the session-scoped bean named `cart`.

```
<c:if
test=
"${sessionScope.cart.numberOfItems > 0}"
>
...
</c:if>
```

JavaServer Faces technology uses the EL for the following functions:

- Deferred and immediate evaluation of expressions
- The ability to set as well as get data
- The ability to invoke methods

See Using the EL to Reference Backing Beans for more information on how to use the EL in JavaServer Faces applications.

To summarize, the EL provides a way to use simple expressions to perform the following tasks:

- Dynamically read application data stored in JavaBeans components, various data structures, and implicit objects
- Dynamically write data, such as user input into forms, to JavaBeans components
- Invoke arbitrary static and public methods
- Dynamically perform arithmetic operations

The EL is also used to specify the following kinds of expressions that a custom tag attribute will accept:

- Immediate evaluation expressions or deferred evaluation expressions.

An immediate evaluation expression is evaluated at once by the underlying technology, such as JavaServer Faces.

A deferred evaluation expression can be evaluated later by the underlying technology using the EL.

- Value expression or method expression.

A value expression references data, whereas a method expression invokes a method.

- Rvalue expression or lvalue expression.

An rvalue expression can only read a value, whereas an lvalue expression can both read and write that value to an external object.

Finally, the EL provides a pluggable API for resolving expressions so custom resolvers that can handle expressions not already supported by the EL can be implemented.

Immediate and Deferred Evaluation Syntax

The EL supports both immediate and deferred evaluation of expressions.

Immediate evaluation means that the expression is evaluated and the result returned as soon as the page is first rendered.

Deferred evaluation means that the technology using the expression language can use its own machinery to evaluate the expression sometime later during the page's lifecycle, whenever it is appropriate to do so.

Those expressions that are evaluated immediately use the ${ } syntax.

Expressions whose evaluation is deferred use the #{ } syntax.

Because of its multiphase lifecycle, JavaServer Faces technology uses mostly deferred evaluation expressions.

During the lifecycle, component events are handled, data is validated, and other tasks are performed in a particular order.

Therefore, a JavaServer Faces implementation must defer evaluation of expressions until the appropriate point in the lifecycle.

Other technologies using the EL might have different reasons for using deferred expressions.

Immediate Evaluation

All expressions using the `${ }` syntax are evaluated immediately.

These expressions can be used only within template text or as the value of a tag attribute that can accept runtime expressions.

The following example shows a tag whose value attribute references an immediate evaluation expression that gets the total price from the session-scoped bean named cart:

```
<fmt:formatNumber
value="${sessionScope.cart.total}"
/>
```

The JavaServer Faces implementation evaluates the expression `${sessionScope.cart.total}`, converts it, and passes the returned value to the tag handler.

Immediate evaluation expressions are always read-only value expressions.

The preceding example expression cannot set the total price, but instead can only get the total price from the cart bean.

Deferred Evaluation

Deferred evaluation expressions take the form `#{expr}` and can be evaluated at other phases of a page lifecycle as defined by whatever technology is using the expression.

In the case of JavaServer Faces technology, its controller can evaluate the expression at different phases of the lifecycle, depending on how the expression is being used in the page.

The following example shows a JavaServer Faces `inputText` tag, which represents a text field component into which a user enters a value.

The `inputText` tag's `value` attribute references a deferred evaluation expression that points to the `name` property of the `customer` bean:

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

For an initial request of the page containing this tag, the JavaServer Faces implementation evaluates the `#{customer.name}` expression during the render-response phase of the lifecycle.

During this phase, the expression merely accesses the value of `name` from the `customer` bean, as is done in immediate evaluation.

For a postback request, the JavaServer Faces implementation evaluates the expression at different phases of the lifecycle, during which the value is retrieved from the request, validated, and propagated to the `customer` bean.

As shown in this example, deferred evaluation expressions can be

- Value expressions that can be used to both read and write data
- Method expressions

Value expressions (both immediate and deferred) and method expressions are explained in the next section.

Value and Method Expressions

The EL defines two kinds of expressions: value expressions and method expressions.

Value expressions can either yield a value or set a value.

Method expressions reference methods that can be invoked and can return a value.

Value Expressions

Value expressions can be further categorized into rvalue and lvalue expressions.

Rvalue expressions can read data but cannot write it.

Lvalue expressions can both read and write data.

All expressions that are evaluated immediately use the ${ } delimiters and are always rvalue expressions.

Expressions whose evaluation can be deferred use the #{ } delimiters and can act as both rvalue and lvalue expressions.

Consider the following two value expressions:

```
${customer.name}
#{customer.name}
```

The former uses immediate evaluation syntax, whereas the latter uses deferred evaluation syntax.

The first expression accesses the `name` property, gets its value, adds the value to the response, and gets rendered on the page.

The same can happen with the second expression.

However, the tag handler can defer the evaluation of this expression to a later time in the page lifecycle, if the technology using this tag allows.

In the case of JavaServer Faces technology, the latter tag's expression is evaluated immediately during an initial request for the page.

In this case, this expression acts as an rvalue expression.

During a postback request, this expression can be used to set the value of the `name` property with user input.

In this case, the expression acts as an lvalue expression.

Referencing Objects Using Value Expressions

Both rvalue and lvalue expressions can refer to the following objects and their properties or attributes:

- **JavaBeans components**
- **Collections**
- **Java SE enumerated types**
- **Implicit objects**

To refer to these objects, you write an expression using a variable that is the name of the object.

The following expression references a backing bean (a JavaBeans component) called `customer`:

```
${customer}
```

The web container evaluates the variable that appears in an expression by looking up its value according to the behavior of `PageContext.findAttribute(String)`, where the `String` argument is the name of the variable.

For example, when evaluating the expression `${customer}`, the container will look for `customer` in the page, request, session, and application scopes and will return its value.

If `customer` is not found, a null value is returned.

You can use a custom EL resolver to alter the way variables are resolved.

For instance, you can provide an EL resolver that intercepts objects with the name `customer`, so that `${customer}` returns a value in the EL resolver instead.

To reference an `enum` constant with an expression, use a `String` literal.

For example, consider this `Enum` class:

```
public enum Suit
{hearts, spades, diamonds, clubs}
```

To refer to the `Suit` constant `Suit.hearts` with an expression, use the `String` literal `"hearts"`.

Depending on the context, the `String` literal is converted to the `enum` constant automatically.

For example, in the following expression in which `mySuit` is an instance of `Suit`, `"hearts"` is first converted to `Suit.hearts` before it is compared to the instance:

```
${mySuit == "hearts"}
```

Referring to Object Properties Using Value Expressions

To refer to properties of a bean or an `enum` instance, items of a collection, or attributes of an implicit object, you use the `.` or `[]` notation.

To reference the `name` property of the `customer` bean, use either the expression `${customer.name}` or the expression `${customer["name"]}`.

The part inside the brackets is a `String` literal that is the name of the property to reference.

You can use double or single quotes for the `String` literal.

You can also combine the `[]` and `.` notations, as shown here:

```
${customer.address["street"]}
```

Properties of an `enum` constant can also be referenced in this way.

However, as with JavaBeans component properties, the properties of an `Enum` class must follow JavaBeans component conventions.

This means that a property must at least have an accessor method called `get`*`Property`*, where *`Property`* is the name of the property that can be referenced by an expression.

For example, consider an `Enum` class that encapsulates the names of the planets of our galaxy and includes a method to get the mass of a planet.

You can use the following expression to reference the method `getMass` of the `Enum` class `Planet`:

```
${myPlanet.mass}
```

If you are accessing an item in an array or list, you must use either a literal value that can be converted to `int` or the `[]` notation with an `int` and without quotes.

The following examples could resolve to the same item in a list or array, assuming that `socks` can be converted to `int`:

- `${customer.orders[1]}`
- `${customer.orders.socks}`

In contrast, an item in a `Map` can be accessed using a string literal key; no coercion is required:

```
${customer.orders["socks"]}
```

An rvalue expression also refers directly to values that are not objects, such as the result of arithmetic operations and literal values, as shown by these examples:

- `${"literal"}`
- `${customer.age + 20}`
- `${true}`
- `${57}`

The EL defines the following literals:

- Boolean: `true` and `false`
- Integer: as in Java
- Floating-point: as in Java
- String: with single and double quotes; `"` is escaped as `\"`, `'` is escaped as `\'`, and `\` is escaped as `\\`
- Null: `null`

You can also write expressions that perform operations on an `enum` constant.

For example, consider the following `Enum` class:

```
public enum Suit
{club, diamond, heart, spade}
```

After declaring an `enum` constant called `mySuit`, you can write the following expression to test whether `mySuit` is `spade`:

```
${mySuit == "spade"}
```

When it resolves this expression, the EL resolving mechanism will invoke the `valueOf` method of the `Enum` class with the `Suit` class and the `spade` type, as shown here:

```
mySuit.valueOf(Suit.class, "spade"}
```

Where Value Expressions Can Be Used

Value expressions using the `${}` delimiters can be used in

- Static text
- Any standard or custom tag attribute that can accept an expression

The value of an expression in static text is computed and inserted into the current output.

Here is an example of an expression embedded in static text:

```
<some:tag>
    some text ${expr} some text
</some:tag>
```

If the static text appears in a tag body, note that an expression will not be evaluated if the body is declared to be `tagdependent`.

Lvalue expressions can be used only in tag attributes that can accept lvalue expressions.

A tag attribute value using either an rvalue or lvalue expression can be set in the following ways:

- With a single expression construct:

```
<some:tag value="${expr}"/>
<another:tag value="#{expr}"/>
```

These expressions are evaluated, and the result is converted to the attribute's expected type.

- With one or more expressions separated or surrounded by text:

```
<some:tag
value=
"some${expr}${expr}text${expr}"
/>
<another:tag
value=
"some#{expr}#{expr}text#{expr}"
/>
```

These kinds of expression, called composite expressions, are evaluated from left to right.

Each expression embedded in the composite expression is converted to a `String` and then concatenated with any intervening text.

The resulting `String` is then converted to the attribute's expected type.

- With text only:

```
<some:tag value="sometext"/>
```

This expression is called a literal expression.

In this case, the attribute's `String` value is converted to the attribute's expected type.

Literal value expressions have special syntax rules.

See Literal Expressions for more information.

When a tag attribute has an `enum` type, the expression that the attribute uses must be a literal expression.

For example, the tag attribute can use the expression `"hearts"` to mean `Suit.hearts`.

The literal is converted to `Suit`, and the attribute gets the value `Suit.hearts`.

All expressions used to set attribute values are evaluated in the context of an expected type.

If the result of the expression evaluation does not match the expected type exactly, a type conversion will be performed.

For example, the expression `${1.2E4}` provided as the value of an attribute of type `float` will result in the following conversion:

```
Float.valueOf("1.2E4").floatValue()
```

See Section 1.18 of the JavaServer Pages 2.2 Expression Language specification (available from http://jcp.org/aboutJava/communityprocess/final/jsr245/) for the complete type conversion rules.

Method Expressions

Another feature of the EL is its support of deferred method expressions.

A method expression is used to invoke an arbitrary public method of a bean, which can return a result.

In JavaServer Faces technology, a component tag represents a component on a page.

The component tag uses method expressions to invoke methods that perform some processing for the component.

These methods are necessary for handling events that the components generate and for validating component data, as shown in this example:

```
<h:form>
<h:inputText
id="name"
value="#{customer.name}"
validator=
"#{customer.validateName}"
/>
<h:commandButton
id="submit"
action="#{customer.submit}"
/></h:form>
```

The `inputText` tag displays as a text field.

The `validator` attribute of this `inputText` tag references a method, called `validateName`, in the bean, called `customer`.

Because a method can be invoked during different phases of the lifecycle, method expressions must always use the deferred evaluation syntax.

Like lvalue expressions, method expressions can use the `.` and the `[]` operators.

For example, `#{object.method}` is equivalent to `#{object["method"]}`.

The literal inside the `[]` is converted to `String` and is used to find the name of the method that matches it.

Once the method is found, it is invoked, or information about the method is returned.

Method expressions can be used only in tag attributes and only in the following ways:

- With a single expression construct, where *bean* refers to a JavaBeans component and *method* refers to a method of the JavaBeans component:

```
<some:tag value="#{bean.method}"/>
```

The expression is evaluated to a method expression, which is passed to the tag handler.

The method represented by the method expression can then be invoked later.

- With text only:

```
<some:tag value="sometext"/>
```

Method expressions support literals primarily to support `action` attributes in JavaServer Faces technology.

When the method referenced by this method expression is invoked, the method returns the `String` literal, which is then converted to the expected return type, as defined in the tag's tag library descriptor.

Parameterized Method Calls

The EL offers support for parameterized method calls.

Method calls can use parameters without having to use static EL functions.

Both the `.` and `[]` operators can be used for invoking method calls with parameters, as shown in the following expression syntax:

- *`expr-a[expr-b](parameters)`*
- *`expr-a.identifier-b(parameters)`*

In the first expression syntax, *`expr-a`* is evaluated to represent a bean object.

The expression *expr-b* is evaluated and cast to a string that represents a method in the bean represented by *expr-a*.

In the second expression syntax, *expr-a* is evaluated to represent a bean object, and *identifier-b* is a string that represents a method in the bean object.

The *parameters* in parentheses are the arguments for the method invocation.

Parameters can be zero or more values or expressions, separated by commas.

Parameters are supported for both value expressions and method expressions.

In the following example, which is a modified tag from the `guessnumber` application, a random number is provided as an argument rather than from user input to the method call:

```
<h:inputText
value=
"#{userNumberBean.userNumber('5')}"
>
```

The preceding example uses a value expression.

Consider the following example of a JavaServer Faces component tag that uses a method expression:

```
<h:commandButton
action="#{trader.buy}" value="buy"
/>
```

The EL expression `trader.buy` calls the `trader` bean's `buy` method.

You can modify the tag to pass on a parameter.

Here is the revised tag where a parameter is passed:

```
<h:commandButton
action="#{trader.buy('SOMESTOCK')}"
value="buy"
/>
```

In the preceding example, you are passing the string `'SOMESTOCK'` (a stock symbol) as a parameter to the `buy` method.

For more information on the updated EL, see http://uel.java.net/.

Defining a Tag Attribute Type

As explained in the previous section, all kinds of expressions can be used in tag attributes.

Which kind of expression and how it is evaluated, whether immediately or deferred, are determined by the `type` attribute of the tag's definition in the Page Description Language (PDL) that defines the tag.

If you plan to create custom tags, for each tag in the PDL, you need to specify what kind of expression to accept.

Table 6-1 shows the kinds of tag attributes that accept EL expressions, gives examples of expressions they accept, and provides the type definitions of the attributes that must be added to the PDL.

You cannot use #{} syntax for a dynamic attribute, meaning an attribute that accepts dynamically calculated values at runtime.

Similarly, you also cannot use the ${} syntax for a deferred attribute.

Table 6-1 Definitions of Tag Attributes That Accept EL Expressions

Attribute Type	Example Expression	Type Attribute Definition

Dynamic	`"literal"`	`<rtexprvalue>`
		`true`
		`</rtexprvalue>`

	`${literal}`	`<rtexprvalue>`
		`true`
		`</rtexprvalue>`

Deferred value	`"literal"`	`<deferred-value>`
		`<type>`
		`java.lang.String`
		`</type>`
		`</deferred-value>`

	`#{customer.age}`	`<deferred-value>`
		`<type>int</type>`
		`</deferred-value>`

Deferred method	`"literal"`	`<deferred-method>`
		`<method-signature>`
		`java.lang.String submit()`
		`</method-signature>`
		`<deferred-method>`

	`#{customer.`
	`calcTotal}`
		`<deferred-method>`
		`<method-signature>`
		`double calcTotal`
		`(int, double)`
		`</method-signature>`
		`</deferred-method>`

In addition to the tag attribute types shown in Table 6-1, you can define an attribute to accept both dynamic and deferred expressions.

In this case, the tag attribute definition contains both an `rtexprvalue` definition set to `true` and either a `deferred-value` or `deferred-method` definition.

Literal Expressions

A literal expression is evaluated to the text of the expression, which is of type `String`.

A literal expression does not use the `${}` or `#{}` delimiters.

If you have a literal expression that includes the reserved ${} or #{} syntax, you need to escape these characters as follows:

- By creating a composite expression as shown here:

```
${'${'}exprA}
#{'#{'}exprB}
```

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

- By using the escape characters `\$` and `\#` to escape what would otherwise be treated as an eval-expression:

```
\${exprA}
\#{exprB}
```

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

When a literal expression is evaluated, it can be converted to another type.

Table 6-2 shows examples of various literal expressions and their expected types and resulting values.

Table 6-2 Literal Expressions

Expression	Expected Type	Result
`Hi`	`String`	`Hi`
`true`	`Boolean`	`Boolean.TRUE`
`42`	`int`	`42`

Literal expressions can be evaluated immediately or deferred and can be either value or method expressions.

At what point a literal expression is evaluated depends on where it is being used.

If the tag attribute that uses the literal expression is defined to accept a deferred value expression, when referencing a value, the literal expression is evaluated at a point in the lifecycle that is determined by other factors, such as where the expression is being used and to what it is referring.

In the case of a method expression, the method that is referenced is invoked and returns the specified `String` literal.

For example, the `commandButton` tag of the `guessnumber` application uses a literal method expression as a logical outcome to tell the JavaServer Faces navigation system which page to display next.

Operators

In addition to the `.` and `[]` operators discussed in Value and Method Expressions, the EL provides the following operators, which can be used in rvalue expressions only:

- Arithmetic: `+`, `-` (binary), `*`, `/` and `div`, `%` and `mod`, `-` (unary)

- Logical: `and`, `&&`, `or`, `||`, `not`, `!`

- Relational: `==`, `eq`, `!=`, `ne`, `<`, `lt`, `>`, `gt`, `<=`, `ge`, `>=`, `le`.

Comparisons can be made against other values or against Boolean, string, integer, or floating-point literals.

- **Empty:** The `empty` operator is a prefix operation that can be used to determine whether a value is `null` or empty.

- **Conditional:** `A ? B : C`.

  Evaluate `B` or `C`, depending on the result of the evaluation of `A`.

The precedence of operators highest to lowest, left to right is as follows:

- `[]` `.`
- `()` (used to change the precedence of operators)
- `-` (unary) `not` `!` `empty`
- `*` `/` `div` `%` `mod`
- `+` `-` (binary)
- `<` `>` `<=` `>=` `lt` `gt` `le` `ge`
- `==` `!=` `eq` `ne`
- `&&` `and`
- `||` `or`
- `?` `:`

Reserved Words

The following words are reserved for the EL and should not be used as identifiers:

`and`	`or`	`not`	`eq`
`ne`	`lt`	`gt`	`le`
`ge`	`true`	`false`	`null`
`instanceof`	`empty`	`div`	`mod`

Examples of EL Expressions

Table 6-3 contains example EL expressions and the result of evaluating them.

Table 6-3 Example Expressions

EL Expression	Result
`${1 > (4/2)}`	false
`${4.0 >= 3}`	true
`${100.0 == 100}`	true
`${(10*10) ne 100}`	false
`${'a' < 'b'}`	true

`${'hip' gt 'hit'}`	`false`
`${4 > 3}`	`true`
`${1.2E4 + 1.4}`	`12001.4`
`${3 div 4}`	`0.75`
`${10 mod 4}`	`2`
`${!empty param.Add}`	`False` if the request parameter named `Add` is `null` or an empty string.
`${pageContext.request.contextPath}`	The context path.
`${sessionScope.cart.numberOfItems}`	The value of the `numberOfItems` property of the session-scoped attribute named `cart`.

`${param['mycom.productId']}`	The value of the request parameter named `mycom.productId`.
`${header["host"]}`	The host.
`${departments[deptName]}`	The value of the entry named `deptName` in the `departments` map.
`${requestScope['javax.servlet.forward.servlet_path']}`	The value of the request-scoped attribute named `javax.servlet.forward.servlet_path`.

`#{customer.lName}`	Gets the value of the property `lName` from the `customer` bean during an initial request. Sets the value of `lName` during a postback.
`#{customer.calcTotal}`	The return value of the method `calcTotal` of the `customer` bean.