DZone Refcardz

CONTENTS INCLUDE:

Essential JSP Expression Language

By Bear Bibeault

ABOUT JSP EXPRESSION LANGUAGE

The JavaServer Pages (JSP) Expression Language (EL) is a simple non-procedural scripting language that can be used to evaluate dynamic expressions within a JSP page. Each EL expression evaluates to a single value that is then expressed as text in the output of the JSP (when used in template text), or passed as a value to a JSP action.

As such, it is ideal for adding dynamic elements to the HTML page (or other text output) generated by the execution of a JSP.

The EL—in concert with the JSP Standard Tag Library (JSTL) —is intended to supplant the need for Java (in the form of scriptlets and scriptlet expressions) in JSP pages, resulting in JSPs that are pure templates rather than unwieldy and error-plagued procedural components.

The EL syntax was inspired by the JavaScript (ECMAScript, to be pedantic) expression syntax. So those familiar with JavaScript should find the syntax familiar.

This refcard focuses on the Expression Language as applied to JSP pages. The EL described here is a subset of the fuller Unified Expression Language that applies to not only JSP pages, but also to JavaServer Faces (JSF) pages. To keep this refcard focused on JSP, JSF-only aspects have not been included.

USING THE EL IN A JSP PAGE

An EL expression is delimited in a JSP page using the notation:

```
${el-expression-goes-here}
```

When the JSP is executed, the value of the expression is determined and the delimiters, along with the enclosed expression, is replaced with the text evaluation of the result.

Note If you want the sequence `${` to appear as template text in your JSP page for some reason, escape the $ character with the backslash (\) character, as in: `\${`.

For example, the EL expression `${ 3 + 4 }` will be replaced with the text 7 in the response output or attribute value.

Using the EL in a JSP Page , continued

Hot Tip The `${ }` delimiters enclose the entire EL expression.

Some people mistakenly think of the `${ }` as a sort of "fetch" operator that must enclose individual elements. Not so!

For example, `${ a + b }` is correct (we'll see later what a and b represent), while `${ ${a} + ${b} }` is not.

Another example that we might find in the template text of a JSP (don't worry about the details, we'll be getting to that soon enough):

```
<p>There are ${thing.count} thing${(thing.count==1)
? '':'s'} available.</p>
```

When executed, if the value of the expression `thing.count` evaluates to the value 3 (again, don't worry about how yet), the following text would be placed into the response buffer:

```
<p>There are 3 things available.</p>
```

LITERAL VALUES

In the previous section, we saw the use of a number of numeric and text literals. Literals represent fixed values such as `3, 4, 's'` and `''` (the empty string).

Literal Values, continued

Numeric literals can be expressed as integer values such as 3 or 213, or as floating point values such as 2.058 and 0.9999. Exponential notation such as 1.23E5 can also be used. Integer literals are, by far, the most commonly used.

Text literals are delimited using either the single-quote character (') or the double-quote character("). The same character must be used to delimit the beginning and the end of the literal. For example: `"Hi there!"` or `'Hello'`.

If the text literal must contain the quote character that is being used as the delimiter, the character must be escaped with the backslash character (\). For example:

```
${'He said, "My name is O\'Hara."'}
```

The backslash character itself must be expressed as `\\`.

There are two more types of literals: the Boolean literals, which consist of the values `true` and `false`, and the Null Literal consisting of the value `null`.

Hot Tip

Be aware that when an EL expression is used within the value of the attribute to a standard or custom action, quoting gets complicated as the attribute value itself is quoted using one of the single or double-quote characters.

For example, the JSTL action:

```
<c:out value="${"Don't look back!"}"/>
```

will result in a JSP exception as the double quote characters in the EL expression interfere with the quoting of the attribute value. The correct syntax would be:

```
<c:out value="${'Don\\'t look back!'}"/>
```

Note that the backslash itself must be escaped so that it "survives" both levels of interpretation (EL and attribute value).

Luckily, when using the EL in template text, these quoting issues rarely raise their head.

INTRODUCING SCOPED VARIABLES

Fixed values are all well and good, but the true strength of the EL lies in expressing dynamic values. These values can be generated from a number of sources, but are always represented by ***scoped variables***.

The concept of a ***variable*** should be familiar: a named element that represents a dynamic value that can be assigned to it. But the concept of the JSP ***scopes*** may be new.

These scopes define the order in which variable names are resolved, the lifetime of the variable, and its purview. The scopes, in their default search order, are: `page`, `request`, `session` and `application`.

In Java code, these scoped variables are created by the `setAttribute()` method of their respective Java classes. These classes (or interfaces), as well as the other properties of the scopes are shown in the following table:

Introducing Scoped Variables, continued

scope	Java class/interface	purview	lifetime
page	PageContext	Current JSP page	Execution of the current JSP page.
request	ServletRequest	Current request	Lifetime of the current request.
session	HttpSession	Resources participating in the active session	Lifetime of the active session.
application	ServletContext	All resources in the web application.	Lifetime of the application context.

Scoped variables can be created in many places, but a very common scenario is for the page controller servlet to create scoped variables in request scope, then forward to a JSP page in order to display that data.

For example:

```
request.setAttribute("greeting","Hi there!");
```

This statement creates a request-scoped variable named `greeting` with the string value `Hi there!`.

Now let's see how we can use that variable in a JSP.

Hot Tip

Application scope is a very handy place to store information that needs to be made available to all resources within a web application. It's easy to establish such data whenever a web application starts up by doing so in ***context listener***.

A context listener is simply a Java class that implements the `javax.servlet.ServletContextListener` interface, and is declared as a listener in the deployment descriptor (`web.xml`) using the `<listener>` element.

Methods in such a class are invoked whenever a web application is put into (and taken out of) service, making it an ideal choice for application setup (and tear down).

And best of all, any scoped variables placed in application scope are available to any EL expression in any JSP in the application!

REFERENCING SIMPLE VALUES

The example code in the previous section created a scoped variable named `greeting` that contains a simple text value. Regardless of what kind of value a scoped variable contains, it can be referenced in an EL expression simply by naming it. For example:

```
${greeting}
```

When evaluated, this EL expression will result in the text `Hi there!` being emitted into the response output stream.

Scoped variables can be referenced anywhere in an EL expression. Let's say that the scoped variables `a` and `b` contain numeric values. If we wanted to add the values together and emit the result, we could write:

```
${a + b}
```

Referencing Simple Values, continued

When a scoped variable is referenced, it is searched for in the order shown in the previous table: first in page scope, then in request scope, then in session scope, and finally in application scope.

Hot Tip Regardless of what type of values are referenced, the final evaluated result is always converted to its string equivalent before being emitted.

So if the result is a Java object whose class has no explicit `toString()` method defined, you might end with goop like the following in the output:

```
org.bibeault.dzone.MyBean@af100d
```

This can be rectified by making sure that the class (in this case, MyBean) has an appropriate `toString()` method defined.

GRABBING JAVABEAN PROPERTIES

While strings and numbers are useful for representing scalar values, more frequently, complex objects in the form of JavaBeans are what we have to deal with, so the EL is specifically designed to make it easy to access the properties of JavaBeans.

For example, imagine a JavaBean of class Person that has properties such as: firstName, lastName, address, and title. These would be represented in the Java class respectively with the accessor methods: `getFirstName()`, `getLastName()`, `getAddress()` and `getTitle()`.

If an instance of this class were to be created as a scoped variable named `person`, we could use the EL's ***property operator*** to reference the property values. This operator has two forms; the simplest and most commonly-used form is the dot (period) character. For example, to reference the bean's title property:

```
${person.title}
```

Similarly, we could reference the firstName property with:

```
${person.firstName}.
```

This operator can be chained. Imagine that the address property is itself a JavaBean whose class defines a number of properties, one of which is city. We could access the person's city value with:

```
${person.address.city}
```

The more general form of the property operator consists of square bracket characters(`[]`) which contain an expression whose string evaluation is taken as the property to be referenced. Consider:

```
${person['title']}
```

This expression is identical to `${person.title}`. Since it's more complex, and slightly more difficult to read, you might wonder "Why bother?"

Grabbing JavaBean Properties, continued

Imagine that the name of the property to be accessed is *itself* in a scoped variable, let's say named `someProperty`. You could access the property using:

```
${person[someProperty]}
```

We'll see some other cases where the more general notation must be used.

Hot Tip Scoped variables that represent classes which do not conform to the JavaBean rules are generally much less useful as EL targets. By design, the EL has no means to call general methods of Java classes, or to access data that is not available as JavaBean properties.

This is one of the reasons that non-bean classes like database result sets are ill-suited for use in JSPs. (There are a bazillion other good reasons that database classes should not be propagated out of the persistence layer, but that's another subject!)

One of the major duties assigned to page controllers is ensuring that the data being sent to a JSP conforms to proper JavaBean standards so that the data can be easily consumed by the EL.

In other words, keep the controllers smart, and the JSP pages stupid.

DEALING WITH COLLECTIONS

In addition to scalar values and beans, the EL can easily reference the elements of certain Java collection constructs. The EL allows us to access the elements of Java arrays, and of collections that implement the `java.util.List` interface, or the `java.util.Map` interface.

Elements within Java arrays and List implementors can be accessed via the indexing operator which just happens to also be the square bracket characters. (Whether the square brackets are interpreted as the property operator or the indexing operator depends upon what type of data it is operating upon.)

The expression within the square brackets must evaluate to an integer numeric value that is used as the zero-based index into the collection. So, assuming that scoped variable `list` is an array or List, you could use any of:

```
${list[0]}
${list[3+4]}
${list[a]}
${list[a+b]}
```

to access elements of the collection as long as the expression within the brackets evaluates to an integer value that is within the range of the collection. (How the EL deals with out-of-bounds values depends upon how it is being used in an expression, but generally, the reference will either be nulled or zeroed. An exception is usually not thrown unless the index expression is not numeric.)

Dealing with Collections, continued

When the scoped variable is an instance of a class that implements `java.util.Map`, the same property operator that is used with JavaBeans is employed to access the map entries. But rather than the enclosed expression evaluating to the name of the property to be fetched, the expression value is taken as the key of the Map entry to be referenced.

So if scoped variable `myMap` is a Map instance that contains an entry with the key `fred`, the EL expressions:

```
${myMap.fred}
${myMap['fred']}
```

could be used to reference the value of the Map entry.

In essence, (at least as far as the EL is concerned) you can think of a Map as a bean with dynamic properties.

ARITHMETIC, RELATIONAL, LOGICAL AND OTHER OPERATORS

In the previous sections, we briefly saw a few operators being used within EL expressions: the addition (+) operator, and even a glimpse of the ternary (?) operator.

In actuality, there are quite a number of operators that can be used within EL expressions. We'll start by looking at the arithmetic operators.

operator	meaning
+	addition
-	subtraction, unary minus
*	multiplication
/ or div	division
% or mod	modulus (remainder)

Examples:

```
<p>3 + 4 = ${3 + 4}</p>
<div>2 cubed is ${2 * 2 * 2}
The remainder of ${a} over ${b} is ${a mod b}
```

The relational operators are:

operator	meaning
eq or ==	equals
ne or !=	not equals
lt or <	less than
le or <=	less than or equals
gt or >	greater than
ge or >=	greater than or equals

Examples:

```
<c:if test="${a == b}">${a} is equal to ${b}</c:if>
<p>True or false? ${a} is less than ${b} is ${a < b}</p>
```

The logical operators are:

operator	meaning
and or &&	and
or or ||	or
not or !	not

Arithmetic, Relational, Logical and Other Operators, continued

Examples:

```
<c:if test="${a==b and c==d}">It's true!</c:if>
<c:if test="”${not a}>It's not true!</c:if>
```

A special operator lets us test if a scoped variable exists or if a collection has no elements, the *empty* operator:

operator	meaning
empty	A unary operator that evaluates to `true` if: ■ the operand is null, or ■ the operand is an empty string, or ■ the operand is an empty array, Map or List Otherwise, evaluates to `false`.

Examples:

```
<c:if test="”${empty a}”>The scoped variable a is empty!</c:if>
<c:if test="”${empty myCollection}”>The collection is empty!</c:if>
<c:if test="”${not empty myCollection}”>The collection is not empty!</c:if>
<c:if test="”${empty myString}”>The string is empty!</c:if>
<c:if test="”${empty myBean.someProperty}”>The property is empty!</c:if>
```

Another special operator is the *ternary* operator, also know as the *conditional* operator:

operator	meaning
?	An operator that evaluates to one of two conditional expressions. The format is: `conditional ? expression1 : expression2` If the conditional expression evaluates to true, then the value of the operation is expression1, else expression2.

Examples:

```
<p>The switch is ${switchState ? 'on' : 'off'}.</p>
<p>The value is ${(value % 2 == 1) ? 'odd' : 'even'}.</p>
<label>The value:</label> ${(empty value) ? 'N/A' : value}
<p>There are ${thing.count} thing${(thing.count==1) ? '':'s'} available.</p>
```

The precedence of all these operators is as follows:

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

Parentheses can be used to affect evaluation precedence in the customary manner.

USING EL FUNCTIONS

Although accessing general methods of scoped variables is not possible with the EL, the EL does make provisions for defining functions that *can* be invoked as part of an EL expression.

These functions have some severe limitations, the most stringent being that the implementation of the function must be a static method of a Java class.

EL functions are defined within a TLD (Tag Library Definition) file, alongside custom tags, using XML syntax that defines the class and method that implements the function, as well as the function signature.

A typical TLD entry might be:

```
<function>
   <name>toJSON</name>
   <function-class> org.bibeault.dzone.refcardz.
      CoreELFunctions </function-class>
   <function-signature> java.lang.String toJSON(
      java.lang.Object ) </function-signature>
</function>
```

This entry defines an EL function named `toJSON` which accepts any Java object as an argument, and returns a String value. The function is implemented by a static Java method named `toJSON` (it doesn't have to match, but it's conventional) in a class named CoreELFunctions. The Java signature of this method is:

```
public static String toJSON( Object value )
```

Note Because this is a static function, it has no implicit access to the environment of the JSP. If your function needs such access, it's customary to pass the `pageContext` implicit variable (see next section) as the first parameter to the method.

The EL functions are accessed using the same notation used in most other languages: name the function and provide any parameters in parentheses. But because functions are defined in a TLD, the namespace of the TLD must be specified.

For example, if a function named `fred` was defined in a TLD mapped to namespace `xyz`, it might be called as follows:

```
${xyz:fred(a,b,c)}
```

Hot Tip The JSP Standard Tag Library (JSTL) defines a handful of useful EL functions using the namespace `fn`. These functions are primarily focused on text processing and give access to many of the most useful methods defined on the java.lang.String class.

Additionally, it provides the `length()` function which returns the length of a String instance, or the size of a collection. For example, if `list` is a scoped variable that contains a collection:

```
${fn:length(list)}
```

will evaluate to the number of elements in the collection.

THE JSP IMPLICIT VARIABLES

In order to allow the EL to interact with the environment of a JSP page above and beyond scoped variables that are defined by the web developer, a number of useful implicit scoped variables are pre-defined that can be used in any EL expression on a JSP page.

These implicit variables are:

implicit variable	description
`pageContext`	The PageContext instance for the JSP page. This bean provides important properties that allow the EL to access many critical environment values (such as the request instance and so on).
`pageScope`	A Map of all scoped variables in page scope.
`requestScope`	A Map of all scoped variables in request scope.
`sessionScope`	A Map of all scoped variables in session scope.
`applicationScope`	A Map of all scoped variables in application scope.
`param`	A Map of all request parameters on the current request. Only the first of any values for a request parameter is mapped as a String.
`paramValues`	A Map of all request parameters on the current request. All the values for a request parameter are mapped as a String array.
`header`	A Map of all request headers for the current request. Only the first of any values for a request header is mapped as a String.
`headerValues`	A Map of all request headers for the current request. All the values for a request header are mapped as a String array.
`cookie`	A Map of each available cookie mapped to a Cookie instance.
`initParam`	A Map of all context parameter values defined in the deployment descriptor. (Not to be confused with servlet init parameters!)

Examples:

The submitted parameter xyz is `${param.xyz}`

The value of the xyz cookie is `${cookie.xyz}`

The xyz context parameter: `${initParam.xyz}`

The context path is `${pageContext.request.contextPath}`

The request URI is `${pageContext.request.requestUri}`

Hot Tip We saw earlier that when you reference a scoped variable by name, let's say `fred`, all scopes, beginning with page scope, are searched for `fred`.

If you want to limit the search to a single scope, you can use one of the implicit scoped Map variables to short-circuit that search. For example:

```
${sessionScope.fred}
```

will only find scoped variables named `fred` in session scope because the implicit scoped variable `sessionScope` only contains entries for session-scoped variables.

Note that if a scoped variable named `fred` exists in a higher-precedence scope (such as page or request scope) this is the only way that you can access `fred` in session scope.

WRAP-UP

Hopefully this refcard has provided a good kick-start in wrapping your mind around the JSP Expression Language and serves as a useful quick reference.

For more details, particularly the manner in which error conditions are handled, please refer to:

JavaServer Pages Specification Version 2.0 (Section JSP.2.2)
Expression Language Specification Version 2.1

The former is actually an easier read if you are only interested in using the EL on JSP pages rather than on JavaServer Faces (JSF) pages.

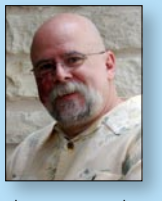
Hot Tip

One final note: the ability for the JSP Expression Language to easily access java.util.Map entries is a much more important and extensible ability than might be apparent at first. Essentially, this allows you to treat maps as beans with dynamic properties. Also, classes that implement the Map interface can create some incredibly flexible functionality.

For two examples of using The EL and maps in a pliable manner, please see the following articles published in issues of the JavaRanch Journal:

The Power of the Map and The Constant Constants Consternation

ABOUT THE AUTHOR



Bear Bibeault

Bear Bibeault has been writing software for over three decades, starting with a Tic-Tac-Toe program written on a Control Data Cyber supercomputer via a 100-baud teletype. He is a Software Architect for a company that builds applications used by the IT administrators that high-performance data centers keep shackled to their servers. He also serves as a "sheriff" at the popular JavaRanch.com.

Books published

- jQuery in Action
- Ajax in Practice
- Prototype and Scriptaculous in Action

RECOMMENDED BOOK

Head First Servlets & JSP will show you how to write servlets and JSPs, what makes the Container tick, how to use the new JSP Expression Language (EL), and much more.

BUY NOW
books.dzone.com/books/hfservlets-jsp

Get More FREE Refcardz. Visit refcardz.com now!

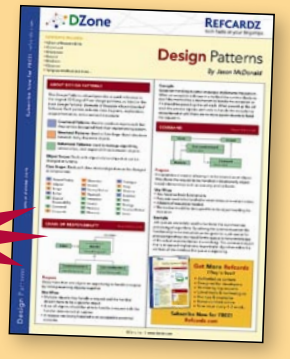
Upcoming Refcardz:

Core CSS: Part III
Using XML in Java
Core Mule 2
Getting Started with Equinox OSGi
SOA Patterns
EMF

Available:

Getting Started with Hibernate Search
Core Seam
Essential Ruby
Essential MySQL
JUnit and EasyMock
Spring Annotations
Getting Started with MyEclipse
Core Java
Core CSS: Part II
PHP
Getting Started with JPA
JavaServer Faces
Core CSS: Part I
Struts2
Core .NET
Very First Steps in Flex
C#
Groovy
NetBeans IDE 6.1 Java Editor
RSS and Atom
GlassFish Application Server

FREE



Design Patterns
Published June 2008

Visit refcardz.com for a complete listing of available Refcardz.

DZone

DZone communities deliver over 4 million pages each month to more than 1.7 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheatsheets, blogs, feature articles, source code and more. "DZone is a developer's dream," says PC Magazine.

DZone, Inc.
1251 NW Maynard
Cary, NC 27513
888.678.0399
919.678.0300
Refcardz Feedback Welcome
refcardz@dzone.com
Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-41-7
ISBN-10: 1-934238-41-4
50795
9 781934 238417
$7.95