

# Expression Languages

A JSP tag, no matter what magic it weaves behind the scenes, is only as powerful and as flexible as the data we are able to pass into it. While some tags are able to make use of the tag body, allowing us to pass in complex, dynamically generated content to the tag for processing, for many tags this is not sufficient, and they rely instead on parameters being passed via the tag's attributes. Yet hard-coded tag attributes don't allow for much flexibility from a tag. The alternative, provided by JSP, is to embed what are known as run-time expressions into the attribute. But run-time expressions, coded in Java, force page designers to contribute Java code to the generated Java servlet, which requires a programming knowledge of the servlet API. It also uses an ungainly syntax that isn't compatible with XML.

This is where the first, and most important, innovation of the JSTL comes in: the expression language (EL).

JSTL will endorse some mechanism for page writers to write plain text **expressions** in certain attributes of the standard tags. Such expressions will be simple strings, which are then interpreted and evaluated at run time by the tags in order to obtain a value. At the time of writing, it was not entirely set in stone what the chosen mechanism would be, but there are three possibilities:

- ❑ The JSTL will incorporate an expression language specification
- ❑ The JSTL will incorporate an expression language extension mechanism and implementations will offer a choice of languages
- ❑ The JSTL will endorse one expression language, and an extension mechanism will be provided to allow substitution of a different language

In a future version of JSP, it is possible that the run-time expression evaluation mechanism will be entirely replaced with a mechanism based on the expression language mechanism used by the JSTL. So there are some important consequences of the decisions that are made now.

If, as seems most likely, the JSPTL specification does incorporate a specification for an expression language, then the question remains, what will that expression language be?

As of the Early Access 1.2 release of the JSTL (JSPTL as it was then), four candidate expression languages have been included in the distribution, along with two mechanisms to select the expression language to be used. The candidates are:

- ❑ SPEL, the Simplest Possible Expression Language
- ❑ JPath, a more powerful language taking some inspiration from XPath
- ❑ JXPath, which is an XPath interpreter that can apply XPaths to object graphs
- ❑ JavaScript/ECMAScript, the well known browser programming language

There's a large variation in terms of the power and complexity of these languages. For a while, SPEL was the only expression language provided, but now that the candidates are all present, with the EA2 release the default expression language was changed to JavaScript, apparently to elicit feedback on its usefulness as an alternative to the very limited SPEL.

These languages are by no means the only possible expression languages. An expression language simply has to be able to evaluate a string supplied by the programmer, returning an object if the evaluation is successful. In fact, there is a fifth expression language, for testing purposes, included in the EA release, called dummy, which simply returns the string JSTL in place of any expression that begins with a dollar-sign (\$). That's obviously not very useful, but the point is that it is entirely up to the expression language to interpret the string it's given, and use it to decide what object to return.

The objects we're most likely to be after are those relating to the state of our web application: objects in the page, request, session and application scopes, strings in the page parameter list, request headers, cookies, and so on. So principally, the job of an expression language is to allow us to specify which of those objects we are interested in.

There are plenty of language styles out there to choose from. It's interesting that relative newcomer XPath has inspired two of the reference implementation candidates; they could equally have drawn on SQL, EJB query language, LDAP distinguished names, or even Java. In this chapter, we'll look at what the driving forces are in the design of an expression language, and examine the candidates to see how they address these issues.

## The Need for an Expression Language

Let's look at how the need for an expression language arose. A simple loop tag, that displays its content *n* times, might look like this:

```
<%@ taglib prefix="my" uri="http://www.wrox.com/ea/made-up-tags" %>
...
<my:loop count="3">
    Hello <br/>
</my:loop>
...
```

Obviously, in this tag, the only way we can tell the loop how many times to run is by passing in the value as an attribute. But what if we want to run the loop a variable number of times, depending on the value of a page parameter? Well, in current JSP pages, we make use of a run-time expression (known as an **RTE**expression):

```
<%@ taglib prefix="my" uri="http://www.wrox.com/ea/made-up-tags" %>
...
<my:loop count="<%=request.getParameter("count")%>">
  Hello <br/>
</my:loop>
...
```

This expression, which is written in Java, is copied directly into the generated servlet, so is executed when the servlet's execution reaches this point in the page. The resulting value is passed in to the custom tag, as if it had been written as a literal value by the page author.

But run-time expressions run counter to the entire philosophy of taglibs, by forcing the page author to be aware of the behind-the-scenes functionality of the generated servlet. In addition, the syntax is ungainly, and breaks XML well-formedness rules, which prevents JSP pages from being treated as pure XML.

It would be better if we could write something more like this:

```
<%@ taglib prefix="my" uri="http://www.wrox.com/ea/made-up-tags" %>
...
<my:loop count="$param:count">
  Hello <br/>
</my:loop>
...
```

This is much simpler, easier for a page author to write, and it's XML-friendly. This is exactly what expression languages are designed to do – in fact, the syntax shown here is that used by SPEL, which will be the first language we'll examine in detail.

Expression language expressions (**ELexpressions**) are intended to fit within the conventional attribute syntax, and, at least at present, are treated just like any other attribute, both during servlet generation, and during page execution. Only when the string is passed into the tag is it evaluated, by the tag itself explicitly obtaining an expression evaluator and requesting the string's evaluation.

In the future, if an expression language is added to the core JSP specification, then tags will receive the results of expressions ready-evaluated – just as they do now from RTexpressions.

## The Dangers of Retrofitting

Introducing a major language feature like an expression language into an established standard like JSP is not without its dangers. Here are three obvious ones:

## Confusion

Obviously, the fact that expression evaluation is handled by tags means that not all tags will support it – in fact, not all of the JSTL tags support expression evaluation. Naturally, this could well lead to a very unpredictable world, where some tags understand expressions, and some don't – or worse, where some attributes on the same tag understand expressions, and others don't. This situation is expected to be resolved by the next version of JSP, which should explicitly incorporate EL support. This will see expression-language evaluation performed by the generated servlet, before it calls in to any custom tags, just like RTexpressions today, and make the expression language equally available to everybody.

Details of how to write tags in the mean time that take advantage of the JSTL expression library are given in Chapter 6, *Collaboration*.

As we go through the tags that exist in the JSTL, we'll point out which attributes can accept EExpressions, and which can't.

## Collision with RTexpressions

Another danger is presented by the fact that EExpressions passed to a tag are simply, from the point of view of the tag library descriptor, strings. We know that we can specify strings as attributes on a tag in two ways – using a literal string, and using an RTexpression. However, if the output of an RTexpression might be passed as input to an expression language evaluator, then there is a serious security issue. If the user can coerce the RTexpression into producing a valid EExpression in place of a simple string, they might be able to access internal page state, or even (depending on the power of the expression language) execute arbitrary code on the server, with the privileges of the JSP page. So, the JSTL tag libraries have been built in such a way that attributes that can accept an EExpression are not allowed to be written with an RTexpression. Problem solved.

This is all very well until you find you need access to a value the expression language you are using can't describe, and want to use an RTexpression instead. To get around this, two separate tag libraries are provided for all the JSTL tag sets, with two separate TLDs. For example, the core tag library (which we'll be discussing in this chapter and the next) has the URI `http://java.sun.com/jstl/ea/core`, but the version of the core library that supports RTexpressions has the URI `http://java.sun.com/jstl/ea/core-rt`.

The RT libraries perform no expression evaluation themselves, so can accept either a string literal or evaluated RTexpression as input. The EL libraries, on the other hand, refuse evaluated RTexpressions, only accepting string literals, which they evaluate themselves.

In the future, if expression evaluation moves to JSP itself, the EL libraries will be dropped, and the RT libraries will be able to accept RTexpressions, EExpressions, or strings.

## Expressions and Values

It's also clear that some mechanism is required to distinguish between a value that the programmer intends to be interpreted as an expression, and a value that is intended as a literal string or other type. The convention that has emerged in some of the candidate expression languages, borrowed loosely from the UNIX shell's in-string escape character, is that expressions begin with a dollar (\$) character. However, not all expression languages support this convention (XPath, notably, doesn't, reserving dollar signs for its own purposes, and treats *all* strings as expressions), and this also clearly could cause some problems where the intended value is a quantity of US, Australian or Canadian currency, for example.

Some ambiguity appears to have emerged in where responsibility lies for interpreting values that are not intended as expressions. The convention adopted, logically enough, by the JSTL reference implementation tags, is to take any attributes for which an EL expression is a valid input, and ask for them to be evaluated. The expression evaluator then takes responsibility for deciding if it can shortcut evaluation by simply returning a literal value. However, the method used to convert the input string into the desired return type (which could be a number, or a date) is not consistent between expression languages. The result is that the output from a simple tag might vary depending on the expression language selected (particularly if XPath is used).

## Expression Language Functionality

So, what functionality can (or should) an expression language expose? To examine this, we'll have to think about exactly what an expression is, at its most basic. To do that, we'll need to standardize on some common terminology.

### Terminology

Some terms, which are common to the grammar of expressions in many different languages, are used in this chapter in accordance with the following definitions:

- ❑ **Expression:** a string of characters that can be interpreted in accordance with an expression language, and **evaluated**. An expression often consists of a combination of several other expressions, with evaluation of the whole expression depending upon the evaluation of sub-expressions. Some typical kinds of expression are **operations**, **identifiers**, **literals**, and **function-calls**.
- ❑ **Evaluation:** a process that takes an **expression** as an input, and produces a **value** as an output.
- ❑ **Value:** a piece of data in a format understood by the computer, such as an object, a string, a date, or a number. We'll follow the Java language's concept of values, and regard the run-time type of a piece of data as being an intrinsic part of its value.
- ❑ **Operation:** an **expression** in which one or more **sub-expressions** (the operands) are operated upon by an operator. For example, an addition operation might take two operands, and evaluate to the sum of the values obtained by evaluating each operand. A **function-call** is a special kind of operation.

- ❑ **Identifier:** an **expression** that uniquely refers to a piece of data by name. The most common kind of identifier we work with in programming is the variable name. An identifier evaluates to the value of the piece of data it refers to. Identifiers can be complex – an identifier referring to a member of an array might be indexed, and the index might itself be an expression. Or we might refer to a property of a variable.
- ❑ **Literal:** an **expression** that consists of an unambiguous string representation of a specific value. For example, the string `-1` might be a literal representation of the number minus one. A literal evaluates to the represented value. A language may understand some, all, or more, of the following types of literal:
  - ❑ String literal
  - ❑ Integer literal
  - ❑ Floating point literal
  - ❑ Boolean literal
- ❑ **Function-call:** an **expression** that consists of a reference to a specific function, and a (possibly empty) set of **expressions**, which are the function's arguments. Function-calls are very similar to **operations**; the function is the operator, and the arguments the operands. However, they have a special status in many expression languages, and there is also the special case of a function call that takes no arguments, in which case it is almost like a special form of **identifier**, typically differing in that the value returned is calculated *at the time the function is called*, not at the time the value was stored.

As we can see, in a typical expression language, an expression can be simple – a single literal or identifier – or can be a complex composite of other expressions – an operation that acts on a literal and an identifier, which is indexed using a function call that takes a literal as an argument. The amount of flexibility a language has in determining what value to return will be defined by what identifiers, functions, operators, and literals it supports. So, what possibilities are there for a language in this area? And what lessons can we learn from other languages about how to tackle the syntax issues?

## Identifiers

As stated above, an identifier can be a lot more complex than simply being a variable name. Let's take a look at some of the ways that an expression language can extend the reach of identifiers.

## Scope

Referring to variables by name is adequate if you are programming in an environment where every possible name can only be allocated to one piece of data. But, to bring us down to earth from the high-level computer science for a moment, we're programming within a JSP page. Now page 1, chapter 1, book 1 of our JSP-101 course text taught us that data within a JSP page context lives in one of four scopes:

- ❑ Page
- ❑ Request
- ❑ Session
- ❑ Application

Now, these scopes are all, effectively, independent namespaces. We can have a piece of data identified by the name `userName` in request scope, and a different piece of data identified by the same name can exist in session scope, at the same time. Obviously, an expression language that only allows us to specify the name of the data in an identifier, without allowing us to specify scope at the same time, is limiting the data we can access. However, The JSP `PageContext` object exposes a method, `findAttribute()`, that performs an ordered search of all of these scopes looking for the attribute requested, and returns the first attribute matching the name that it finds. So this can be used to allow access to all four scopes, but with the danger that attributes in narrower scopes will mask attributes with the same name in wider scopes.

In addition, there is data elsewhere that we may want to access by name. Request parameters, cookies, HTTP request headers, perhaps even objects stored in a J2EE application server's JNDI naming context. A mechanism that allows selection of different scopes could also allow data to be retrieved from all kinds of different contexts. This opportunity, it has to be said, has not been grasped by most of the available expression languages.

### Indexing and Introspection

Having retrieved an object by name, it may turn out that what we're interested in is not the object itself, but one of the object's properties. Or, if the object in question is an array, or collection, we may be interested in accessing a member by numeric index. If it's a map, perhaps we might want to access its members by name. These situations can be catered for by having the language allow for introspection, or indexing, in its identifier syntax.

Introspection allows us to discover what properties an object has. We can, for example, allow the expression programmer to access the values of public, readable properties according to the JavaBeans specification (any method of the form `public a getb()`). An expression language might provide for this using the familiar Java `'.'` member access operator, so we could call the `userName` object's `getFirstName()` method as follows:

```
userName.firstName
```

Obviously, we would want to be able to chain a number of these property accesses together to access deeply nested data:

```
company.financeDepartment.manager.homeAddress.city
```

Indexing should, similarly, allow us to refine down the piece of data we're retrieving:

```
company.department[13].team[2]
```

Again, this is just an example syntax (although one that should feel natural to a Java programmer) – however, the fact that properties are accessed by property name, not via the fully named method, might cause a few bugs while you get used to it.

It might be possible for an expression language to map such an expression to several different object constructs. `company.department[13]` might mean `company.getDepartment()[13]`, `company.getDepartment(13)`, `company.getDepartment().get(13)`, or `company.getDepartment().get(new Integer(13))`. Ideally, it would handle all of these options.

Handling the last option, which would be the syntax needed to retrieve an item from a Map object, would also allow such syntax to be applied to non-numeric indexes, such as string indexes.

### Filters and Predicates

Indexed properties are a hint towards something else. When we ask for the property with index  $n$ , one way of looking at it (the XPath way of looking at it, if we're honest) is to say that we're selecting from the list of available properties, the one whose position is  $n$ . To put it another way, we are applying a filter to the list of properties, that lets through only the property, or properties, we are interested in. The XPath term for such a filter is a predicate, and we'll use the two terms pretty well interchangeably here.

An XPath predicate is a powerful concept. A predicate is an expression, which is applied to each item in a set. Any item whose predicate evaluates to false is dropped from the set; only those for whom the predicate evaluates to true are kept. It provides a similar role in an XPath expression, to a WHERE clause in a SQL expression.

If an expression language had effective support for predicates, we would be able to select data of interest not only on the basis of how it relates to its 'parents', but on the basis of properties of its 'children'. For example, extending the example from above, a predicate-capable expression language might allow us to write an expression like this:

```
company.department[manager.name='Jan']
```

In other words, we would select the department(s) whose manager property has a name property equal to Jan.

As you can probably tell, a predicate-aware expression language gives us a considerable amount of power in identifying the piece of data we're interested in. This sort of expression is more of a query than a simple name, somewhat like using a search engine to find a website, rather than typing in its URL.

## Operators

There are three principal classes of operator of interest in expression programming.

### Boolean Operators

Boolean operators, sometimes called conditional operators, return a Boolean (true or false) value. The classic Boolean operators are:

- ❑ Equality (usually either == or =)
- ❑ Inequality (usually either != or <>. Less frequently, •)
- ❑ Comparison (usually <, >, <=, >=)

Note that the comparison operators differ from the equality and inequality operators, in that generally they can only be applied to numerical values. Equality and inequality operators can generally be applied to a wide range of datatypes, such as strings.

In addition, you may expect there to be Boolean AND and OR operations (possibly symbolized by the words, or && and ||), possibly XOR, and maybe a unary NOT operator (like Java/C/C++'s ! operator). These are less widely supported in the candidate expression languages than you might expect, and the syntax is extremely variable.

### Arithmetic Operators

Arithmetic operators operate on numbers, and return a number. The usual suspects are:

- ❑ Addition (+)
- ❑ Subtraction (-)
- ❑ Multiplication (\*)
- ❑ Division (/)
- ❑ Modulus or remainder (%)

In addition, there's the unary minus operator, which negates the value on its right.

Java, amongst other languages, also has bitwise Boolean operators (&, ^, | and ~) that act on numeric values, but their use is generally fairly specialized. However, they might be of use in an expression language; not essential, but nice to know they're there.

### String Operators

In Java, there is only one String operation, concatenation, using the overloaded + operator. While other operators are possible (perhaps unary case-conversion operators, or regular expression comparisons), generally, additional string functionality is only available through functions. Unless Perl is adopted as an expression language, that's likely to remain the case.

## Other Operations

A sad omission from most of the candidate expression languages is also the classic C/C++/Java ternary conditional operator, the `? :` operator used to choose between two values. In expressions, where multi-line logic isn't available, the ternary operator's terse if/then/else syntax could prove very useful and efficient. The alternative, an Excel/non-standard SQL-style inline-if function, also seems not to have been implemented.

Other operators can also be imagined. Java, for example, considers explicit casts (e.g. `(int)`) to be unary operators. One group of operators common in mathematics has never really made it into programming, and that's set operators. This would include *union*, *intersection*, *is-a-subset-of*, *is-a-superset-of*, *is-an-element-of*, and *contains* operators. The reason I mention these here is that we'll see when we come to the XPath-derived languages that a union operation does exist, although in JXPath's case, its operation on lists is more akin to concatenation than a set-wise union.

## Functions

One way to extend the power of an expression language is to allow the user to call functions. These could be functions defined by the language itself, or defined by the user or some third party, using some language-defined extension mechanism.

One tempting option open to a language running inside a Java environment is to allow the user to use some mechanism to call out to Java methods. This presents difficulties in formulating a syntax that allows unambiguous calls to be made to static methods, as well as instance methods on objects resulting from the evaluation of expressions, but this hasn't stopped people trying.

One danger with the introduction of powerful functionality via functions is the introduction of **side effects**. A side effect is a change in the state of the system, brought about by the execution of a piece of code. For example, when you call the `next()` method of an `Iterator` object, your call has two effects: the next object in the `Iterator` is returned to you, and the `Iterator`'s cursor moves on one item. The `Iterator`'s state has been changed by your call. On the other hand, calling `hasNext()` on the same `Iterator` returns true or false, but does not change the state of the `Iterator` internally. `hasNext()` is side-effect free.

One view of the role of expressions is that they are, effectively, a kind of query. One of the things about querying against data is that if you execute the same query several times, provided the data doesn't change in the mean time, the result is the same. However, if the query operation itself has side effects, then it may alter the data, and executing the same query again will give a different result.

For this reason, it seems logical that we would want to avoid side-effects wherever possible in expressions, so allowing expression writers the freedom to raid the Java class libraries may not be a good thing. However, the ability to cause side effects from expressions may be regarded as essential by some programmers (for example, the kind who write methods that end `'return a++;'`). It certainly allows you to achieve more using simple expressions. But since expressions are only used within attributes of tags, any side effects that do occur will only be, well, side-effects of the real purpose of the expression – to obtain or calculate data to pass in to the tag.

## Expression Language Tags

The first four tags we're going to look at from the JSP standard tag library are all directly related to the use of expression languages. One is used to select an expression language, one to evaluate an expression, and output the result (we'll be using this one a lot), and two to create variables in one scope or another.

First of all, we'll take a look at how the expression language evaluator is chosen for the evaluation of a particular expression.

### Selecting an Expression Language Evaluator

Expression evaluation is performed by classes implementing the `org.apache.taglibs.standard.lang.support.ExpressionEvaluator` interface. One such class is provided for each of the candidate expression languages. Their fully qualified names (which we'll be needing later) are as follows:

Language	Expression Evaluator
Dummy	<code>org.apache.taglibs.standard.lang.dummy.DummyEvaluator</code>
SPEL	<code>org.apache.taglibs.standard.lang.spel.Evaluator</code>
JPath	<code>org.apache.taglibs.standard.lang.jpath.JPathExpressionEvaluator</code>
JXPath	<code>org.apache.taglibs.standard.lang.jxpath.JXPathExpressionEvaluator</code>
JavaScript	<code>org.apache.taglibs.standard.lang.javascript.JavascriptExpressionEvaluator</code>

The web application context parameter `javax.servlet.jsp.jstl.temp.ExpressionEvaluatorClass` is used to specify the default expression language to be used. Context parameters are declared in a `web.xml` file using a `<context-param>` element as a child of the `<web-app>` root element. Here's the element required to specify SPEL as your web application's default expression evaluator.

```
<context-param>
  <param-name>
    javax.servlet.jsp.jstl.temp.ExpressionEvaluatorClass
  </param-name>
  <param-value>
    org.apache.taglibs.standard.lang.spel.Evaluator
  </param-value>
</context-param>
```

Note that the ordering of elements in the `<web-app>` element of a `web.xml` file is significant; all of the `<context-param>` elements in a `web.xml` file must occur after any `<icon>`, `<display-name>`, `<description>`, and `<distributable>` elements, and before any `<filter>` elements.

If no such parameter is specified, the default expression evaluator will, as of EA2 of JSTL, be the JavaScript evaluator.

Within a page, it is possible to override the default expression language selection, simply by using the core JSTL tag, `<c:expressionLanguage>`.

### **<c:expressionLanguage> tag**

```
<c:expressionLanguage evaluator="evaluator_class">
    ...
</c:expressionLanguage>
```

To use the `<c:expressionLanguage>` tag in a JSP page, you must first ensure that the core tag library is declared in the page. The one attribute, `evaluator`, is required, and specifies the fully-qualified name of the evaluator class.

Since this tag's of no use when you're using run-time expressions, there is no version of this tag in the RTexpression version of the core library – only in the EExpression variant.

## **Evaluating Expressions**

The tag that perhaps has the most important role in anchoring expressions into JSP code is the `<c:expr>` tag. It is used simply to output the value of an expression as a string.

### **<c:expr> tag**

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/ea/core" %>
    ...
<c:expr value="EExpression" [ default="default" ]/>
<c:expr value="EExpression"> ... default ... </c:expr>
```

Its one compulsory attribute, `value`, should be an EExpression (although any string that isn't an expression will be passed through as a literal by any well-behaved expression language). An optional second attribute, `default`, is used to provide a value that should be output instead, if an `ExpressionException` is thrown, or if the result of executing the expression is `null`. The value of the `default` attribute can, if required, also be an expression.

As an alternative to using the `default` attribute, a default can be provided by including a tag body. The body is only executed if the expression evaluation throws an `ExpressionException`, or returns `null`.

If there is any danger of the expression not working, or evaluating to `null`, you need to provide a default – if you don't, the tag will fail with an error.

There is no RTexpression version of this tag, because an RTexpression itself can function as a replacement.

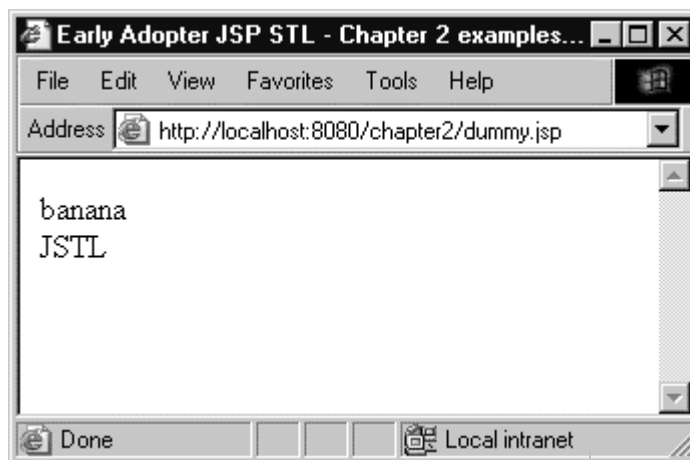
We now (finally) know enough to be able to write a brief page that evaluates an expression. Let's use the dummy expression language for our first experiment:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/ea/core" %>

<html>
  <head>
    <title>
      Early Adopter JSP STL - Chapter 2 examples
    </title>
  </head>
  <body>
    <c:expressionLanguage evaluator=
      "org.apache.taglibs.standard.
        lang.dummy.DummyEvaluator">
      <c:expr value="banana" /> <br/>
      <c:expr value="$banana" /> <br/>
    </c:expressionLanguage>
  </body>
</html>
```

The output looks like this:

Looking at the source code, we can see we declare an expression evaluation scope for the dummy expression language, inside which we evaluate two expressions: banana, and \$banana. As we said before, the dummy evaluator returns JSTL for any expression that begins with a \$ character, and the original expression string otherwise. So, the output is banana for the first expression, and JSTL for the second.



## Variables

Before we proceed with our examination of the expression languages, let's complete our quick tour of core JSTL tags, with a couple of tags that relate to creating and scoping variables.

### <c:set> tag

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/ea/core" %>
...
<c:set var="varName" value="EExpression" [ scope="scope"] />
<c:set var="varName" [ scope="scope"]> ... value ... </c:set>
```

The `<c:set>` tag takes a value (either specified via the `value` attribute, or via element content), and stores it in one of the four scoped page context namespaces – page, request, session or application – as an attribute with the name specified in the `var` attribute.

The `scope` attribute is optional, and can be specified as either `page`, `request`, `session` or `application`. If none is specified, then page scope is assumed.

Like `<c:expr>`, this tag fails with an error if the expression evaluation fails or results in null. This is unfortunate, since sometimes you might actually *want* to set an attribute to null. Similarly, if the content of the tag evaluates to an empty string, the tag will fail, meaning you can't set a variable to an empty string in this way.

Again, there is no RTexpression equivalent, because this tag is directly intended to support EExpression operations.

The non-graceful failure of these two tags is a problem, that requires careful consideration when using them to process user input. User-supplied values are, by definition, unpredictable, and so it is a shame that recovery from unexpected results isn't a higher priority in the design of these most basic tags.

### **<c:declare> tag**

```
<c:declare id="varName" type="qualified_type"/>
```

The `<c:declare>` tag is used to perform a little magic trick to make an attribute stored in the page context available as a scripting variable, and thus accessible by just its name from RTexpressions and scriptlets. This is a tricky act to pull off, since in order for a scripting variable to exist, it has to have a declaration written into the Java of the generated servlet at the time the page is translated. To accomplish this, it makes use of the variable-declaration API exposed to `TagExtraInfo` classes, which is accessed at translation-time to find out what variables need declaring.

All page context attributes can be retrieved using an RTexpression of the form `<%=pageContext.findAttribute("name") %>`. However, one of the most likely reasons for using an RTexpression in a world where EExpressions exist is to allow you to call a method on the returned attribute, and Java will only let you do that if you perform a cast. That makes the RTexpression ugly, unwieldy, and error-prone. Using `<c:declare>`, we can create a safely typed variable for the value, that enables us to call the method with a minimum of fuss. Here's an example, using the `String.toUpperCase()` method. First, let's see how we'd do it without the `<c:declare>` tag:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/ea/core" %>

<html>
  <head>
    <title>
      Early Adopter JSP STL - Chapter 2 examples
    </title>
  </head>
  <body>
```

```
<c:set var="name" value="James Hart" />
<%= ((String)
    pageContext.getAttribute("name")).toUpperCase() %>
</body>
</html>
```

Obviously, that's a pretty ungainly statement. Let's see what we can do about it using `<c:declare>`:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/ea/core" %>

<html>
  <head>
    <title>
      Early Adopter JSP STL - Chapter 2 examples
    </title>
  </head>
  <body>
    <c:set var="name" value="James Hart" />
    <c:declare id="name" type="java.lang.String" />
    <%= name.toUpperCase() %>
  </body>
</html>
```

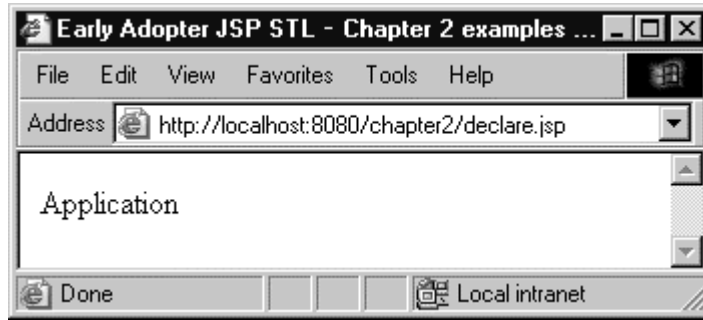
Much tidier, and it's clearer what's going on.

When looking for a page context attribute to map, the `<c:declare>` tag starts off looking in page scope, then proceeds up through request, session and application, until it finds an attribute with the specified name. It then maps the scripting variable to the value of that attribute. We can see that this is what's happening by looking at the following JSP page:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/ea/core" %>

<html>
  <head>
    <title>
      Early Adopter JSP STL - Chapter 2 examples
    </title>
  </head>
  <body>
    <c:set var="name" scope="application"
      value="Application" />
    <c:declare id="name" type="java.lang.String" />
    <c:set var="name" scope="request" value="Request" />
    <c:set var="name" scope="page" value="Page" />
    <%= name %>
  </body>
</html>
```

The output looks like this:



The scripting variable is declared *after* the application scoped attribute is created, but *before* page or request scoped attributes exist. The mapping is thus to the application-scoped value, and the output of the script variable at the end is Application. Move the `<c:declare>` tag until after the page-scoped value has

been created, and the output will be Page. The `<c:declare>` tag is finding an attribute to map from the page context at the time the tag is executed.

Note that you can't declare two scripting variables with the same name in the same page.

Now we've worked with a few tags, we can pop data into the four different page context scopes, and we can also expose it as scripting variables. We're ready to start taking a look at the candidate expression languages.

## Describing Grammars

As we go through the four expression languages, we'll make use of Backus-Naur Form to describe some of the languages in terms of what is called a context-free grammar (we shan't be producing the complete JavaScript expression grammar here, but the other three are small enough to examine in detail).

BNF is relatively simple to understand. A BNF grammar describes the syntax of a language in terms of **terminals**, which are specific strings of characters that have particular meaning in the language. For example, a sequence of digits might constitute a terminal that represents an integer literal; a plus character (+) might constitute a terminal representing the addition operator. A **non-terminal** is a combination of terminals and/or other non-terminals, which together also have a particular meaning in the language. For example, an expression, followed by an addition operator, followed by another expression, might constitute the 'addition expression' non-terminal.

The grammar is written as a series of **productions**; a production is a statement that describes the constitution of a particular non-terminal. For example:

```
number ::= digit+
```

The `::=` symbol can be read as "consists of". The '+' character following the word 'letter' is a quantifier (called a Kleene Plus), which means "one or more times". So, this production says that a `number` consists of one or more `digits`. A precise grammar would treat `digit` as a non-terminal, which needed to be defined as one of a specific set of characters. We won't always need to go into that much detail, preferring human-readable descriptions of such things to interminable lists of character literals.

There is one other quantifier. A '\*' following an entity (a Kleene Star) says that the preceding entity may occur zero or more times. Square brackets around an entity mark it as optional. Literal characters are written within quotation marks. Parentheses ( ( ) ) around a group of entities mark them for treatment as a single entity. A vertical bar ( | ) separates mutually exclusive (either-or) options.

Here's a simple grammar that describes the basic Java properties file format:

```
document ::= line*
line      ::= (property | comment) linebreak
property  ::= key "=" value
comment   ::= "#" text
```

You should be able to see the way the BNF makes clear the possible sequences that constitute a valid document. A document is made up of zero or more lines. A line is either a comment or a property definition, followed by a line break. A property is a key and a value separated by an equals sign. A comment is a hash character followed by some text.

The nature of a context-free grammar is that it unambiguously breaks down the structure of anything that conforms to it, associating each element of it with a defined structure in the grammar.

## SPEL

The **Simplest Possible Expression Language**, SPEL, was originally developed by Nathan Abramson of ATG. It was the first expression language to ship with the first release of the JSTL early access, and its maturity certainly shows, compared to the other languages available, in its consistency and resilience. Its primary design goal is providing strong syntax for accessing data in a variety of contexts, and it also supports Boolean operations, which make it well suited to work with the conditional tags in the core library. It lacks arithmetic or basic string operations, and doesn't support any function or extension syntaxes. It is, as the name implies simple, but that doesn't mean it's not powerful. Let's take a look.

## Grammar

The SPEL grammar consists of the following productions:

```
expr          ::= value | comparison
comparison    ::= value operator value
value         ::= identifier | literal
identifier    ::= [ scope ":" ] (name | string-literal)
               ( "." name ) *
operator      ::= "==" | "!=" | ">" | "<" | ">=" | "<="
scope         ::= "page" | "request" | "session" | "app"
               | "header" | "param" | "paramvalues"
```

SPEL's core grammar is trivial. An expression either consists of a comparison operation, or a single value. Comparison operations operate on values, so the most complex SPEL expression consists of a comparison between two identifiers.

SPEL expressions must begin with the dollar character (\$) to indicate to the SPEL evaluator that the expression needs evaluating. The dollar sign is trimmed off before evaluation begins.

The productions for literals aren't reproduced above, because it's easier to enumerate the literal types supported in a human-readable form by describing them than using BNF. SPEL understands four kinds of literals: string, number, Boolean, and null. Strings are quoted, either with single or double quotes. Numbers can be integers or floating point values, and floating point numbers can be specified using exponent notation (for example,  $2.5e-6$  is "two-point-five times ten to the power of minus six", or 0.0000025). Boolean literals are specified using the words `true` and `false`, and `null` is specified with the word `null`. Whitespace is ignored throughout.

The only other non-terminal production omitted from the above grammar is `name`, because it is more clearly described semantically than syntactically. Let's look a little more closely at identifiers in SPEL.

An identifier points to a specific piece of data. It starts with an optional scope, then a colon (:), then an initial name, or string literal. This is the name of an attribute in the selected scope (or, if no scope is specified, somewhere in the page context, working up through the scopes until a match is found). We'll see why strings are allowed here shortly. Subsequently, there is an optional list of dots followed by names. These are introspective requests for properties. The value returned by the previous part of the identifier is queried to see if it has a readable public JavaBean property matching the given name, and if so, the retrieved value is used. This continues until the end of the identifier is reached.

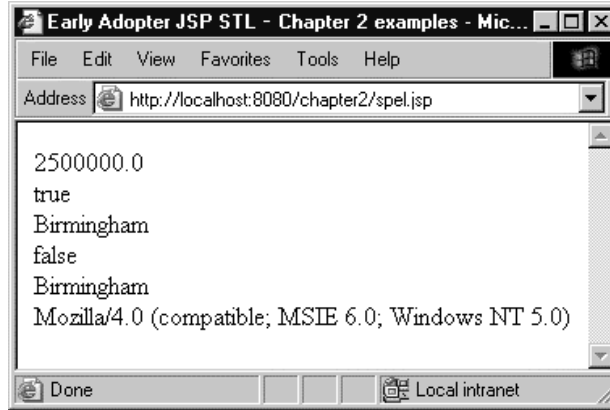
So, let's cast a few quick SPELs to get us going:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/ea/core" %>
<html>
  <head>
    <title>
      Early Adopter JSP STL - Chapter 2 examples
    </title>
  </head>
  <body>
    <c:expressionLanguage evaluator=
      "org.apache.taglibs.standard.lang.spel.Evaluator">
      <c:set scope="request" var="location"
        value="Birmingham"/>
      <c:expr value="$2.5e6"/><br/>
      <c:expr value="$2.5e-6 == 0.0000025"/><br/>
      <c:expr value="$location"/><br/>
      <c:expr value="'birmingham' == location"/><br/>
      <c:expr value="$request:location"/><br/>
    </c:expressionLanguage>
  </body>
</html>
```

```
</c:expressionLanguage>
  <c:expr value="$header:'user-agent'"/><br/>
</body>
</html>
```

The output should look something like this:

After creating a request-scoped parameter, we're evaluating six simple expressions here to see how they're handled:



- ❑ 2.5e6
- ❑ 2.5e-6 == 0.0000025
- ❑ location
- ❑ 'birmingham' == location
- ❑ request:location
- ❑ header:'user-agent'

The first is an example of the literal exponent notation we referred to earlier. After evaluation, we can see that the result is displayed as 2500000.0, showing that the expression language has correctly interpreted it as a number. The second checks our earlier assertion that 2.5e-6 is 0.0000025. This is an example of a comparison operation between two numeric literals.

The next expression performs an unscoped lookup of the name location. This returns the value we stored earlier. Expression four shows a string literal, and a string comparison with an identifier. It also shows that string comparison is case sensitive. The fifth expression is a scoped lookup of the location string.

The final expression, header:'user-agent', is a more interesting example, and makes use of a scoped string-literal, functioning as an identifier. This identifier looks in the header 'scope', for an attribute called user-agent – in other words, it looks for the User-agent: HTTP header (which should be present on any HTTP request). The reason this has to be quoted as a string literal is that SPEL only allows valid Java names as name specifiers. Hyphen characters are not allowed in Java names, so a secondary mechanism, allowing for names that aren't normal Java identifiers, is required, and is provided.

SPEL has a wide choice of scopes. It's able to query for parameters and headers, as well as the usual page context scopes. If no scope is specified, then SPEL looks up the specified name first in page, then request, then session, and finally application context. The parameter scopes are worth noting, particularly because there are two of them.

A request for a named attribute under `param` scope will return the first parameter matching the specified name. A request for an attribute under `paramvalues` scope will return a `java.util.Enumeration`, containing all of the parameters matching the name (there is nothing to stop any HTTP request containing multiple parameters with the same name – pages with groups of checkboxes often return parameters in this form; in fact an HTML combo-box set to allow multiple selections will return multiple matching parameters as a matter of course).

If a value is requested and it doesn't exist in the specified scope, then the expression will fail with an error. This is a shame, since it makes it impossible to simply test if a parameter with a particular name exists.

We'll see, when we come to look at the `<c:forEach>` tag in the next chapter, how an expression that returns an `Enumeration` can be very useful.

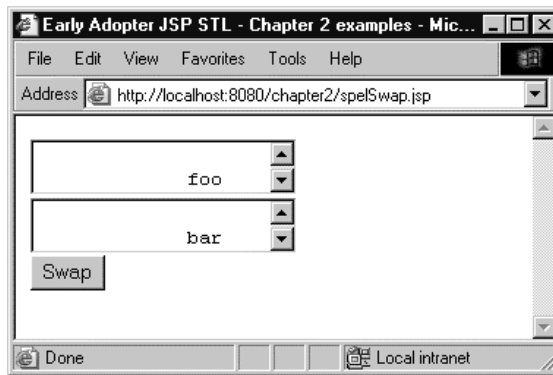
Let's look at how we can use the `param` scope in a page:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/ea/core" %>

<html>
  <head>
    <title>Early Adopter JSP STL - Chapter 2 examples</title>
  </head>
  <body>
    <c:expressionLanguage evaluator=
      "org.apache.taglibs.standard.lang.spel.Evaluator">
      <form method="GET" action="SPEL.jsp">
        <textarea name="word1">
          <c:expr value="$param:word2">
            foo
          </c:expr></textarea>
        <textarea name="word2">
          <c:expr value="$param:word1">
            bar
          </c:expr>
        </textarea><br/>
        <input type="submit" value="Swap"/>
      </form>
    </c:expressionLanguage>
  </body>
</html>
```

The output should look something like this:

This simple HTML form takes two words from the user, and swaps them over (it is, admittedly, unlikely that this is the killer app the web has been waiting for – unfortunately, more powerful examples will have to wait until we've looked at a few more tags). I've used text area controls, as they allow for cleaner code – using text input boxes forces us to place the `<c:expr>` tags inside the `value` attribute of the HTML `<input>` tag, which is valid JSP, but not pretty.



Notice that if there is no parameter matching either of the names we specify (such as is bound to be the case the first time our page is accessed), we provide sensible default values. Also, as you may want to verify yourself by changing the method attribute of the form, it's perfectly possible to use exactly the same code to handle POST requests, since the underlying `ServletRequest.getParameter()` method handling things behind the scenes transparently switches between looking up POST data and looking up values from the query string when handling GET requests.

## Limitations

One of the biggest problems SPEL suffers from is that it is limited to basic Boolean operators. A few simple arithmetic operations would make a world of difference to SPEL's utility as an expression language; as it is, it's sometimes frustratingly limiting.

In addition, its introspection capabilities only extend to bean properties. It can't index into Maps by name, or even into arrays or collections by numerical index. It also can't handle indexed properties. Again, this is frustrating at times, and severely limits what SPEL can do at this time.

**!** *However, array support is currently being added to SPEL at the time of writing this book.*

## Advantages

On the other hand, SPEL is well written, performance-optimized (it makes heavy use of object caching), very stable, extremely predictable, easy to learn, and well documented (something that certainly can't be said for some other expression languages). It also provides access to attributes in namespaces other than the page context, to a far greater extent than other languages do.

SPEL is designed to be expandable in the future, so it may be that some additional functionality could be added without it losing its status as a simple expression language. SPEL certainly shouldn't be dismissed as just a 'test' expression language.



**Expression Languages**