Pro JSP 2

Fourth Edition

Simon Brown, Sam Dalton, Daniel Jepp,
David Johnson, Sing Li, and Matt Raible
Edited by Kevin Mukhar

Apress®

Pro JSP 2, Fourth Edition

Copyright © 2005 by Simon Brown, Sam Dalton, Daniel Jepp, Dave Johnson, Sing Li, and Matt Raible

ISBN (pbk): 1-59059-513-0

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editors: Steve Anglin and Kevin Mukhar
Technical Reviewers: Scott Davis and Dilip Thomas
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis,
  Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Associate Publisher: Grace Wong
Project Manager: Beckie Brand
Copy Edit Manager: Nicole LeClerc
Copy Editor: Sharon Wilkey
Assistant Production Director: Kari Brooks-Copony
Production Editor: Ellie Fountain
Compositor: ContentWorks
Proofreader: Sue Boshers
Indexer: Julie Grady
Artist: Kinetic Publishing Services, LLC
Interior Designer: Van Winkle Design Group
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 3

The JavaServer Pages Expression Language

One of the features of the JSP specification that you'll be using most often is the **JSP expression language**, an intentionally simple language that is, to a large extent, independent from JSP.

In previous incarnations of JSP, Java code was embedded into JSP pages in the form of scriptlets, for example:

```
<%
  MyBean bean = new MyBean();
  String name = bean.getName();
  out.println(name);
%>
```

This scriptlet creates a new instance of a class called `MyBean`, gets its `name` property, assigns this to a string variable, and then outputs this string to the page. Now you might be looking at this and thinking, "I can achieve the same thing by using the JSP standard actions (`<useBean>` and `<getProperty>`)."

Although this is certainly true, it was previously extremely hard to write a function-rich JSP-based web application without using a number of scriptlets within your pages. In fact, there are many problems associated with using Java code in the form of scriptlets in JSP pages. The first and most obvious of these is that it's very common for non-Java programmers to create the user interface for a system. This is because graphic designers are generally better than Java programmers at creating functional user interfaces. The second problem caused by the use of scriptlets is that of maintainability. Embedding large amounts of code into the user interface of a system makes the interface much harder to change and understand.

For all of these reasons, the JSP 2.0 specification introduced an expression language (EL) that can do pretty much everything that scriptlets can do. This language is far simpler to understand than Java and looks very similar to JavaScript. The following are good reasons for this similarity:

- JavaScript is something that most page authors are already familiar with.
- The EL is inspired by ECMAScript, which is the standardized version of JavaScript.

In fact, both ECMAScript and the XPath EL inspired the JSP EL. The EL specification states, ". . . the experts involved were very reluctant to design yet another expression language and tried to use each of these languages, but they fell short in different areas."

If you've been following the progress of JSP, and the JSP Standard Tag Library (JSTL), you're probably aware that the first expression language was released as part of the JSTL. The EL was then incorporated into the JSP 2.0 specification with JSTL 1.1.

At around the same time, the JavaServer Faces (JSF) expert group was developing an expression language for JSF. Because of JSF requirements, the JSF expression language had some differences from the JSP expression language. JSP 2.1 unifies the two versions so that there is a single expression language used for JSP, JSTL, and JSF.

In this chapter, you'll learn the following:

- The syntax and usage of the EL, including reserved words, disabling scriptlets in a page, and disabling the evaluation of the EL on a page or set of pages
- The operators within the EL, including arithmetic operators, comparison operators, logical operators, and other operators
- The use of JavaBeans with the EL
- The implicit objects within the EL
- The declaration and use of functions in the EL

The Syntax and Use of the Expression Language

In this section, you'll look at the syntax of the EL, see how to use it on a JSP page, and learn the reserved words of the language. After you've looked at the basics, you'll move on to look at how and why you might disable the EL and Java scriptlets within a page or set of pages.

Basic Syntax

No matter where the EL is used, it's always invoked in a consistent manner, via the construct `${expr}` or `#{expr}`, where `expr` is the EL expression that you wish to have evaluated.

In the EL 2.1 specification, the syntax of `${expr}` and `#{expr}` are equivalent and can be used interchangeably. However, when used with some other Java Platform, Enterprise Edition API, the other API may enforce restrictions on the use of `${expr}` and `#{expr}`. Specifically, when used with JSP pages, the two forms cannot be used interchangeably. Within a JSP page, `${expr}` is used for expressions that are evaluated immediately, whereas `#{expr}` is used for expressions for which evaluation is deferred. Deferred expressions are used with custom actions, which you will look at in Chapters 6 and 7.

A simple use of the EL is shown here. This piece of code creates a JavaBean and outputs its `name` property:

```
<jsp:useBean id="bean" class="MyBean"/>
${bean.name}
```

We'll discuss the detailed syntax of JavaBeans later in the "JavaBeans and the Expression Language" section.

Note that in the previous example you used the `<useBean>` standard action to create the object. This is the recommended way to do this, rather than instantiating the object in a scriptlet.

Literals

Just as in any programming language, the EL provides several literals for developers to use. A literal can be of a Boolean, integer, floating point, string, or null type. The following are valid values for each literal type:

- **Boolean:** `true` or `false`.
- **Integer**: This is limited to values defined by the `IntegerLiteral` regular expression as follows:

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

  This regular expression says that an integer is any sequence of digits using the digits from 0 to 9. The specification also allows an integer literal to be preceded by a unary "-" symbol to form negative integer literals. For example, the following are valid integers:

  ```
  -102
   0
   21
   21234
  ```

- **Floating point**: This is defined by the following `FloatingPointLiteral` expression:

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

  This expression is more complex. As with integer literals, a floating-point literal can be preceded by a unary "-" symbol to produce negative floating-point literals. To help you understand this, here are some valid floating-point literals:

  ```
  -1.09
  -1.003
   1.0E10
   1.
  -10.0
   0.1
  ```

- **String**: A string is any sequence of characters delimited with either single or double quotes. For example, `"a string"` and `'a string'` are both valid; however, `"as'` and `'as"` are not valid. If you want to represent quotes within a string, then you can use `\"` for double quotes, or `\'` for single quotes. Alternately, if the string is delimited by double quotes, you can use single quotes within the string without escaping the single quotes, and vice versa. To represent a \ in a string, you use the escape sequence `\\`.
- **Null**: You can represent null by using the literal `null`.

Default Values and the Expression Language

Experience suggests that it's most important to be able to provide as good a presentation as possible, even when there are simple errors in the page. To meet this requirement, the EL does not provide warnings, just "default values" and "errors." Default values are type-correct values that are assigned to a subexpression when there is a problem, and errors are exceptions to be thrown (and then handled by the standard JSP error-handling process). An example of such a default value is `'infinity'`. This value is assigned to an expression that results in a divide by zero. For example, the following piece of EL will display infinity rather than causing an error:

```
${2/0}
```

The equivalent Java expression would throw an `ArithmeticException`.

Using the Expression Language

You can use the EL in the same places as you would have used a scriptlet, for example:

- Within attribute values for JSP standard and custom tags
- Within template text (that is, in the body of the page)

Using the Expression Language Within Custom Tags

Using the EL within the attributes of a custom tag in a JSP page allows you to dynamically specify the attribute values for a custom tag. This is an extremely powerful mechanism. The following code snippet shows how you might dynamically specify an attribute to a custom tag by using a scriptlet:

```
<myTagLibrary:myTag counter="<%= 1+1 %>" />
```

To achieve this dynamic behavior prior to the JSP 2.0 specification, you had to use scriptlets. As we've discussed, scriptlets are untidy and cause all sorts of problems with readability and maintainability. By using an EL statement, you can dynamically provide the values to a custom tag's attribute. If you were to repeat the previous tag example by using the EL, you would see that the code is much neater:

```
<myTagLibrary:myTag counter="${1+1}" />
```

You can see that the value `1+1` is being passed to the custom tag as an attribute named `counter`. The details of the creation of custom tags are discussed at length in Chapters 6 through 8.

You'll look at more advanced use of the language with JavaBeans, arithmetic, and comparisons later in this chapter.

Using the Expression Language Within JSP Template Text

Now that you've seen how the EL can be used to provide the values of custom tag attributes, you'll learn how you can use the EL within the body of a JSP page so that you can produce dynamic content. Listing 3-1 shows an example of a JSP page with some dynamic content generated by the EL. This page displays the value of a parameter (passed to the page) called name. The user is then given a text field in which to enter a new name, and a button to submit the name back to the page for another greeting.

Listing 3-1. *templateText.jsp*

```
<html>
  <head>
    <title>EL and Template Text</title>
    <style>
      body, td {font-family:verdana;font-size:10pt;}
    </style>
  <head>
  <body>
    <h2>EL and Template Text</h2>
    <table border="1">
      <tr>
        <td colspan="2">Hello ${param['name']}</td>
      </tr>
      <tr>
        <form action="templateText.jsp" method="post">
          <td><input type="text" name="name"></td>
          <td><input type="submit"></td>
        </form>
      </tr>
    </table>
  </body>
</html>
```

To run this example, you need to deploy it into a JSP 2.0– or JSP 2.1–compliant web container. As with all examples in this book, we will be using Tomcat 5.5, so you'll need to create the deployment descriptor shown in Listing 3-2.

Listing 3-2. *web.xml*

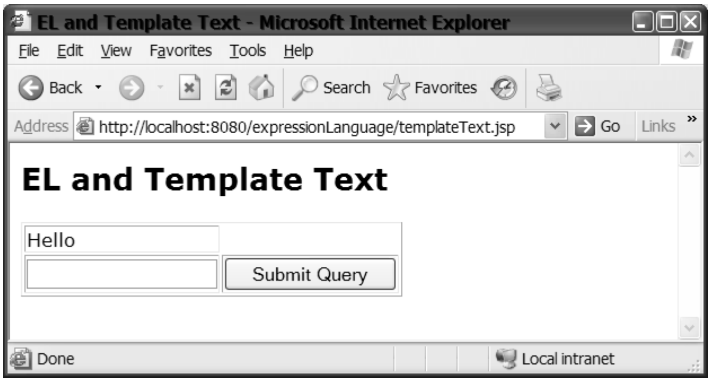
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation= ➥
"http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
         version="2.5">

</web-app>
```

Here is the complete list of steps needed to create, deploy, and run this example:

1. Create the directory `%TOMCAT_HOME%\webapps\expressionLanguage\WEB-INF`.
2. Create the `web.xml` file shown in Listing 3-2. Save it to the `webapps\expressionLanguage\WEB-INF` folder.

3. Create the JSP page in Listing 3-1 and save it to the `webapps\expressionLanguage` folder.
4. Start Tomcat, if needed, open your web browser, and go to `http://localhost:8080/expressionLanguage/templateText.jsp`.

Figure 3-1 shows the page that should appear in the web browser.


EL and Template Text - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:8080/expressionLanguage/templateText.jsp
Go Links
Submit Query
Done
Local intranet

EL and Template Text

Hello

Figure 3-1. *The templateText.jsp displays the value submitted by the user.*

As you can see, this page is a very simple, personalized greeting. When the page first loads, there will be no request parameter, so the greeting will be only the word "Hello." When the Submit Query button is clicked, the request is submitted with the parameter name. The JSP page accesses this parameter and uses an EL statement to print the greeting. You'll look at how the request variable is accessed later, in the "Expression-Language Implicit Objects" section. For now, try entering different values within the text box and clicking Submit Query.

Reserved Words

As with any other language, the JSP EL has many words that are reserved. A reserved word (also known as a keyword) is one that has a special meaning within the language. This means that you cannot use the reserved word to represent anything else, such as a variable identifier. The following are reserved words in the JSP EL:

```
and

eq

gt

true

instanceof

or

ne

lt

false
```

```
empty

not

if

ge

null

div

mod
```

It's worth noting that not all of these words are currently in the language, but they may be in the future, and developers should avoid using them. You'll see examples of using the majority of the reserved words during the course of this chapter.

Disabling Scriptlets

As we've mentioned, the EL is intended to replace the use of Java scriptlets in developing JSP-based web applications. To this end, it's possible to disable the evaluation of scriptlets through configuration parameters. This allows a developer to ensure that no one inadvertently uses scriptlets instead of the EL. This can allow best practices to be more easily enforced.

You can disable scriptlets within a page using the `web.xml` deployment descriptor by choosing to disable evaluation for a single page, a set of pages, or for the entire application.

The tags that you need to add to the deployment descriptor are within the `<jsp-config>` element. The following example disables scriptlets for all JSP pages within an application:

```
<jsp-config>
  <jsp-property-group>
    <url-pattern>*.jsp</url-pattern>
    <scripting-invalid>true</scripting-invalid>
  </jsp-property-group>
</jsp-config>
```

The `<url-pattern>` element can represent a single page, for example:

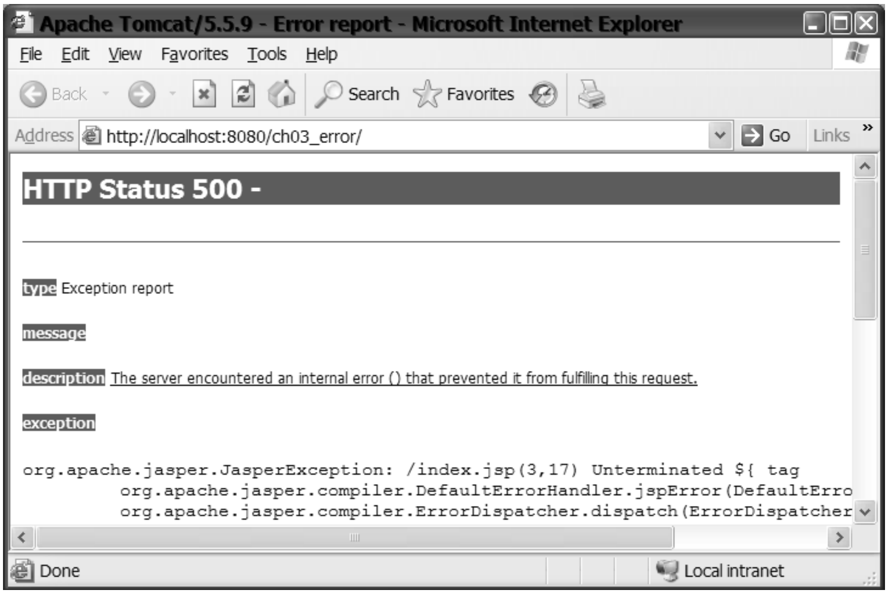
```
<jsp-config>
  <jsp-property-group>
    <url-pattern>/test.jsp</url-pattern>
    <scripting-invalid>true</scripting-invalid>
  </jsp-property-group>
</jsp-config>
```

It can also represent a set of pages, for example:

```
<jsp-config>
  <jsp-property-group>
    <url-pattern>/noscriptlets/</url-pattern>
    <scripting-invalid>true</scripting-invalid>
  </jsp-property-group>
</jsp-config>
```

Disabling the Expression Language

Just as you can disable scriptlets within a page, you can also disable the evaluation of the EL. In previous versions of JSP, the characters `${` had no special meaning; therefore, it's possible that people have used them in their JSP pages. If you were to try to deploy these pages on a JSP 2.0– or JSP 2.1–compliant web container, you would get errors. Figure 3-2 shows the kind of error that you could expect to see.


Apache Tomcat/5.5.9 - Error report - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:8080/ch03_error/ Go Links
Done Local intranet

HTTP Status 500 -

type Exception report

message

description The server encountered an internal error () that prevented it from fulfilling this request.

exception

```
org.apache.jasper.JasperException: /index.jsp(3,17) Unterminated ${ tag
        org.apache.jasper.compiler.DefaultErrorHandler.jspError(DefaultErro
        org.apache.jasper.compiler.ErrorDispatcher.dispatch(ErrorDispatcher
```

Figure 3-2. *A page implemented under an earlier JSP specification may use the characters ${, resulting in an error when the same page is used in a JSP 2.0 or later container.*

It's worth noting that if a web application is deployed by using a Servlet 2.3 deployment descriptor (that is, one that conforms to the 2.3 Document Type Definition, `http://java.sun.com/dtd/web-app_2_3.dtd`), then the evaluation of the EL is automatically deactivated. This is to reduce the chance that an error will occur when a web container is upgraded. Conversely, if a web application is deployed with a Servlet 2.4 or Servlet 2.5 deployment descriptor (that is, the web application conforms to the 2.4 XML Schema, `http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd`, or to the 2.5 XML Schema, `http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd`), then the EL is enabled by default.

You can disable EL evaluation in two ways:

- Individually on each page by using the `page` directive
- Within the `web.xml` file by using a JSP configuration element

To disable the EL for a single page, it's simplest to use the `isELIgnored` attribute of the `page` directive in the header of the page:

```
<%@ page isELIgnored="true" %>
```

If you choose to disable evaluation within the web.xml file, you can disable for a single page, a set of pages, or the entire application. The following XML example shows how you might disable the EL for an entire application:

```
<jsp-property-group>
  <url-pattern>*.jsp</url-pattern>
  <el-ignored>true</el-ignored>
</jsp-property-group>
```

To disable the evaluation of the EL for a single page within the deployment descriptor, you can use an XML fragment similar to the following:

```
<jsp-property-group>
  <url-pattern>noel.jsp</url-pattern>
  <el-ignored>true</el-ignored>
</jsp-property-group>
```

To disable the evaluation of the EL for a set of pages within the deployment descriptor, you can use an XML fragment similar to the following:

```
<jsp-property-group>
  <url-pattern>/noel/</url-pattern>
  <el-ignored>true</el-ignored>
</jsp-property-group>
```

If you try to use the EL after you've disabled it, you won't see any errors and the ${expr} expression within the JSP page will appear unaltered in the final output.

Arithmetic Evaluation Using the Expression Language

Now that you've examined the basic syntax of the EL and where it can be used, you'll look at some specific uses of the language. The first of these is using the EL to evaluate arithmetic operations. There are many cases within web-application development where you need to perform some mathematics on a page. This might be to show a number within the text of a page or to pass a number to a custom tag. In either case, the concepts are exactly the same.

Arithmetic operators are provided to act on both integer and floating-point values. There are six operators that you can use and combine to achieve the vast majority of mathematical calculations with ease:

- Addition: +
- Subtraction: -
- Multiplication: *
- Exponents: E
- Division: / or div
- Modulus: % or mod

The last two operators are presented with two alternative syntaxes (both will produce exactly the same result). This is so that the EL is consistent with both the XPath and ECMAScript syntaxes. You can use all the operators in a binary fashion (that is, with two arguments, such as `2 + 3`) and the subtraction operator to represent the unary minus (that is, `-4 + -2`).

As you would expect, each operator has a precedence that determines the order of evaluation of an expression. This precedence is as follows:

- `()`
- `-` (unary)
- `* / div mod %`
- `+ -` (binary)

You'll update this list when you look at the comparison operators in the next section. You can, of course, use parentheses to change the order of evaluation, as these take the highest precedence.

With operators of equal precedence, the expression is evaluated from left to right, for example:

```
2 * 5 mod 3
```

is equivalent to `(2 * 5) mod 3`, which evaluates to `1`—rather than `2 * (5 mod 3)`, which evaluates to `4`.

Listing 3-3 is a JSP page that shows an example of all the operators in action.

Listing 3-3. *arithmetic.jsp*

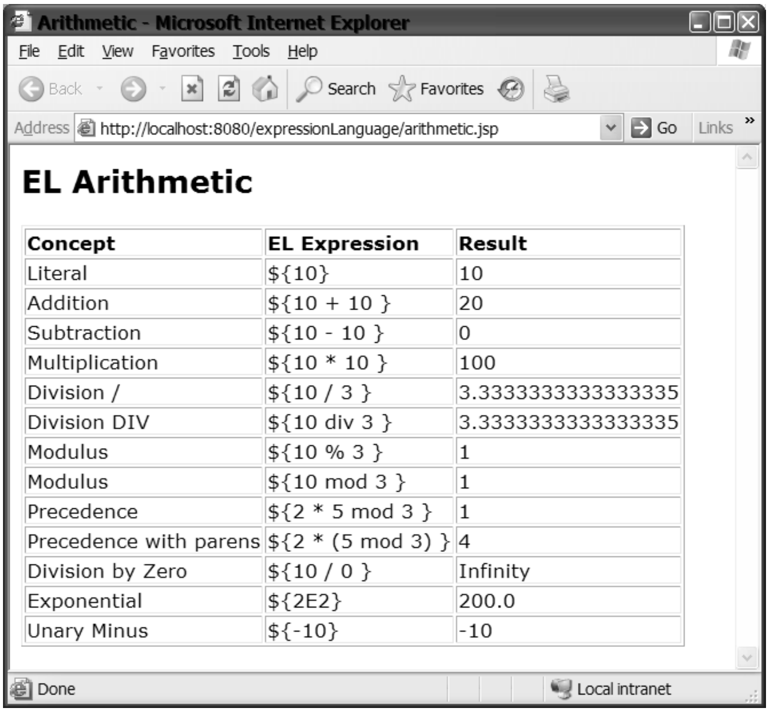
```
<html>
<head>
  <title>Arithmetic</title>
  <style>
    body, td {font-family:verdana;font-size:10pt;}
  </style>
</head>
<body>
  <h2>EL Arithmetic</h2>
  <table border="1">
    <tr>
      <td><b>Concept</b></td>
      <td><b>EL Expression</b></td>
      <td><b>Result</b></td>
    </tr>
    <tr>
      <td>Literal</td>
      <td>${'${'}10}</td>
      <td>${10}</td>
    </tr>
    <tr>
      <td>Addition</td>
      <td>${'${'}10 + 10 }</td>
      <td>${10 + 10}</td>
    </tr>
    <tr>
      <td>Subtraction</td>
      <td>${'${'}10 - 10 }</td>
      <td>${10 - 10}</td>
    </tr>
    <tr>
      <td>Multiplication</td>
      <td>${'${'}10 * 10 }</td>
      <td>${10 * 10}</td>
    </tr>
    <tr>
      <td>Division / </td>
      <td>${'${'}10 / 3 }</td>
      <td>${10 / 3}</td>
    </tr>
    <tr>
      <td>Division DIV</td>
      <td>${'${'}10 div 3 }</td>
      <td>${10 div 3}</td>
    </tr>
    <tr>
      <td>Modulus</td>
      <td>${'${'}10 % 3 }</td>
      <td>${10 % 3}</td>
    </tr>
    <tr>
      <td>Modulus</td>
      <td>${'${'}10 mod 3 }</td>
      <td>${10 mod 3}</td>
    </tr>
    <tr>
      <td>Precedence</td>
      <td>${'${'}2 * 5 mod 3 }</td>
      <td>${2 * 5 mod 3}</td>
    </tr>
    <tr>
      <td>Precedence with parens</td>
      <td>${'${'}2 * (5 mod 3) }</td>
      <td>${2 * (5 mod 3)}</td>
    </tr>
    <tr>
      <td>Division by Zero</td>
      <td>${'${'}10 / 0 }</td>
      <td>${10 / 0}</td>
    </tr>
```

```
    <tr>
      <td>Exponential</td>
      <td>${'${'}2E2}</td>
      <td>${2E2}</td>
    </tr>
    <tr>
      <td>Unary Minus</td>
      <td>${'${'}-10}</td>
      <td>${-10}</td>
    </tr>
  </table>
</body>
</html>
```

To run this example, you need to deploy it into a JSP 2.0– or JSP 2.1–compliant web container just as you did earlier:

1. Copy the JSP page in Listing 3-3 to the `%TOMCAT_HOME%\webapps\expressionLanguage` directory.
2. Start Tomcat, if needed, open your web browser, and go to `http://localhost:8080/expressionLanguage/arithmetic.jsp`.

Figure 3-3 shows the output of the JSP page.


Arithmetic - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:8080/expressionLanguage/arithmetic.jsp
Go Links
Done
Local intranet

EL Arithmetic

Concept	EL Expression	Result
Literal	${10}	10
Addition	${10 + 10 }	20
Subtraction	${10 - 10 }	0
Multiplication	${10 * 10 }	100
Division /	${10 / 3 }	3.3333333333333335
Division DIV	${10 div 3 }	3.3333333333333335
Modulus	${10 % 3 }	1
Modulus	${10 mod 3 }	1
Precedence	${2 * 5 mod 3 }	1
Precedence with parens	${2 * (5 mod 3) }	4
Division by Zero	${10 / 0 }	Infinity
Exponential	${2E2}	200.0
Unary Minus	${-10}	-10

Figure 3-3. *The arithemetic operators allow you to perform many basic math operations in a JSP page.*

All that this JSP page does is print out the result of the expression next to the expression itself. It also demonstrates an interesting technique: that of displaying the `${` characters on a JSP page. This is easily achieved by embedding the literal `'${'` in an EL statement. For example, to show the string `${2+3}` on a JSP page, you can use the following expression:

```
${'${'}2 + 3 }
```

You may be thinking that the operators that are provided are not powerful enough. For example, where is the square-root operator? Advanced operators are deliberately not provided to the JSP developer because advanced calculations should not be done in a page. They should either be done in the controller layer of the application, or by using view-helper components such as custom tags or EL functions (see the "Expression-Language Functions" section later in this chapter).

Comparisons in the Expression Language

Another useful feature of the EL is the ability to perform comparisons, either between numbers or objects. This feature is used primarily for the values of custom tag attributes, but can equally be used to write out the result of a comparison (`true` or `false`) to the JSP page. The EL provides the following comparison operators:

- `==` or `eq`
- `!=` or `ne`
- `<` or `lt`
- `>` or `gt`
- `<=` or `le`
- `>=` or `ge`

The second version of each operator exists to avoid having to use entity references in JSP XML syntax; however, the behavior of the operators is the same.

In the JSP page in Listing 3-4, you can see the comparison operators in use.

Listing 3-4. *conditions.jsp*

```
<html>
<head>
  <title>EL Conditions</title>
  <style>
    body, td {font-family:verdana;font-size:10pt;}
  </style>
</head>
<body>
  <h2>EL Conditions</h2>
  <table border="1">
    <tr>
      <td><b>Concept</b></td>
      <td><b>EL Condition</b></td>
      <td><b>Result</b></td>
    </tr>
    <tr>
      <td>Numeric less than</td>
      <td>${'${'}1 &lt; 2}</td>
      <td>${1 < 2}</td>
    </tr>
    <tr>
      <td>Numeric greater than</td>
      <td>${'${'}1 &gt; 2}</td>
      <td>${1 > 2}</td>
    </tr>
    <tr>
      <td>Numeric less than</td>
      <td>${'${'}1 lt 2}</td>
      <td>${1 lt 2}</td>
    </tr>
    <tr>
      <td>Numeric greater than</td>
      <td>${'${'}1 gt 2}</td>
      <td>${1 gt 2}</td>
    </tr>
    <tr>
      <td>Numeric Greater than or equal</td>
      <td>${'${'}1 &gt;= 1}</td>
      <td>${1 >= 1}</td>
    </tr>
    <tr>
      <td>Numeric Less than or equal</td>
      <td>${'${'}1 &lt;= 1}</td>
      <td>${1 <= 1}</td>
    </tr>
    <tr>
      <td>Numeric less than or equal</td>
      <td>${'${'}1 le 1}</td>
      <td>${1 le 1}</td>
    </tr>
    <tr>
      <td>Numeric greater than or equal</td>
      <td>${'${'}1 ge 1}</td>
      <td>${1 ge 1}</td>
    </tr>
    <tr>
      <td>Numeric equal to</td>
```

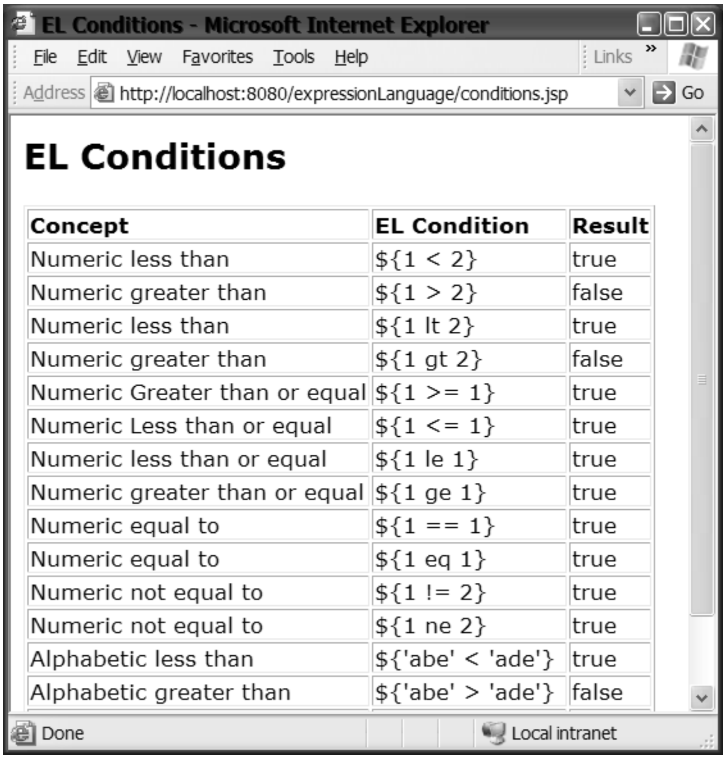
```
      <td>${'${'}1 == 1}</td>
      <td>${1 == 1}</td>
    </tr>
    <tr>
      <td>Numeric equal to</td>
      <td>${'${'}1 eq 1}</td>
      <td>${1 eq 1}</td>
    </tr>
    <tr>
      <td>Numeric not equal to</td>
      <td>${'${'}1 != 2}</td>
      <td>${1 != 2}</td>
    </tr>
    <tr>
      <td>Numeric not equal to</td>
      <td>${'${'}1 ne 2}</td>
      <td>${1 ne 2}</td>
    </tr>
    <tr>
      <td>Alphabetic less than</td>
      <td>${'${'}'abe' &lt; 'ade'}</td>
      <td>${'abe' < 'ade'}</td>
    </tr>
    <tr>
      <td>Alphabetic greater than</td>
      <td>${'${'}'abe' &gt; 'ade'}</td>
      <td>${'abe' > 'ade'}</td>
    </tr>
    <tr>
      <td>Alphabetic equal to</td>
      <td>${'${'}'abe' eq 'abe'}</td>
      <td>${'abe' eq 'abe'}</td>
    </tr>
    <tr>
      <td>Alphabetic not equal to</td>
      <td>${'${'}'abe' ne 'ade'}</td>
      <td>${'abe' ne 'ade'}</td>
    </tr>
  </table>
  </body>
</html>
```

Again, you'll deploy the JSP page into a JSP 2.0– or JSP 2.1–compliant web container. Here is the list of steps to create, deploy, and run the JSP page:

1. Enter the JSP code in Listing 3-4 into a file called `conditions.jsp` and save it to the `expressionLanguage` folder.

2. Start Tomcat, open your web browser, and go to `http://localhost:8080/expressionLanguage/conditions.jsp`.

Figure 3-4 shows the web page generated by this JSP page.


EL Conditions - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Links
Address http://localhost:8080/expressionLanguage/conditions.jsp
Go
Done
Local intranet

EL Conditions

Concept	EL Condition	Result
Numeric less than	${1 < 2}	true
Numeric greater than	${1 > 2}	false
Numeric less than	${1 lt 2}	true
Numeric greater than	${1 gt 2}	false
Numeric Greater than or equal	${1 >= 1}	true
Numeric Less than or equal	${1 <= 1}	true
Numeric less than or equal	${1 le 1}	true
Numeric greater than or equal	${1 ge 1}	true
Numeric equal to	${1 == 1}	true
Numeric equal to	${1 eq 1}	true
Numeric not equal to	${1 != 2}	true
Numeric not equal to	${1 ne 2}	true
Alphabetic less than	${'abe' < 'ade'}	true
Alphabetic greater than	${'abe' > 'ade'}	false

Figure 3-4. *Conditional operators can be used to compare operators in a JSP page.*

You are now in a position to update your precedence table from the previous section with the comparison operators. Again, parentheses can be used to alter the order of evaluation, and identical precedence operators are evaluated from left to right as follows:

- ()
- - (unary)
- * / `div` `mod` %
- + - (binary)
- < > <= >= `lt` `gt` `le` `ge`
- == != `eq` `ne`

Logical Operators in the Expression Language

The EL also enables you to perform logical operations on Boolean arguments. The logical operators are as follows:

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

Once again, there are alternatives for each of the operators.

If either of the arguments is not Boolean, an attempt will be made to convert them to Boolean; if this is not possible, an error will occur.

Listing 3-5 shows some examples of the logical operators in action.

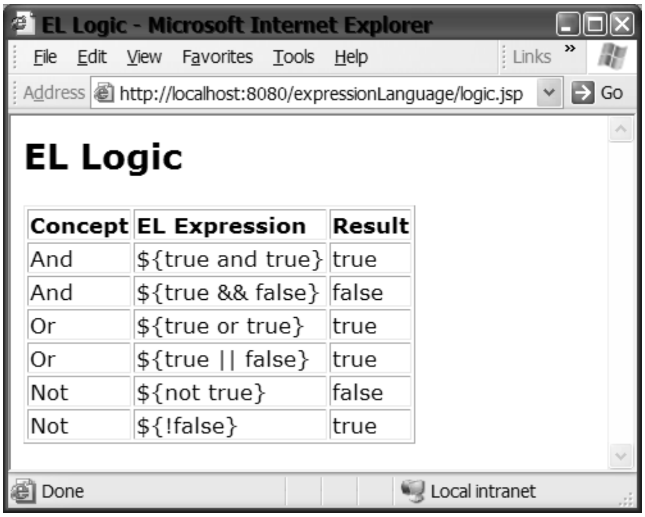
Listing 3-5. *logic.jsp*

```
<html>
<head>
  <title>EL Logic</title>
  <style>
    body, td {font-family:verdana;font-size:10pt;}
  </style>
</head>
<body>
  <h2>EL Logic</h2>
  <table border="1">
    <tr>
      <td><b>Concept</b></td>
      <td><b>EL Expression</b></td>
      <td><b>Result</b></td>
    </tr>
    <tr>
      <td>And</td>
      <td>${'${'}true and true}</td>
      <td>${true and true}</td>
    </tr>
    <tr>
      <td>And</td>
      <td>${'${'}true && false}</td>
      <td>${true && false}</td>
    </tr>
    <tr>
      <td>Or</td>
      <td>${'${'}true or true}</td>
      <td>${true or false}</td>
    </tr>
    <tr>
      <td>Or</td>
      <td>${'${'}true || false}</td>
      <td>${true || false}</td>
    </tr>
    <tr>
      <td>Not</td>
      <td>${'${'}not true}</td>
      <td>${not true}</td>
    </tr>
    <tr>
      <td>Not</td>
      <td>${'${'}!false}</td>
      <td>${!false}</td>
    </tr>
  </table>
</body>
</html>
```

Once again, you'll run this example by deploying it in a web container:

1. Enter the JSP code in Listing 3-5 into a file called `logic.jsp` and save it to the `expressionLanguage` folder.
2. Start Tomcat and go to `http://localhost:8080/expressionLanguage/logic.jsp`.

Figure 3-5 shows the web page generated by this JSP page.


EL Logic - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Links
Address http://localhost:8080/expressionLanguage/logic.jsp
Go
Done
Local intranet

EL Logic

Concept	EL Expression	Result
And	${true and true}	true
And	${true && false}	false
Or	${true or true}	true
Or	${true || false}	true
Not	${not true}	false
Not	${!false}	true

Figure 3-5. *Logical operators can be used with Boolean arguments in a JSP page.*

Other Operators

Besides the arithmetic, comparison, and logical operators that you've seen so far, several other operators are available to developers using the EL. These operators are mainly related to objects. A property or method of an object can be accessed by using either the `.` (dot) or `[]` (bracket) operator, which is deliberate in order to align the language with ECMAScript. For example, `obj.property` is equivalent to `obj["property"]`.

This syntax is also used to access the items within maps, lists, or arrays. For example, to access the item in a map with the key `"sam"`, you could use `myMap["sam"]` or `myMap.sam`.

The final operator that you'll look at is the `empty` operator. The `empty` operator is a prefix operator that can be used to determine if a value is null or empty. To evaluate the expression `empty A`, the EL implementation code first checks whether A is null. If A is null, the expression evaluates to `true`. If A is an empty string, array, map, or an empty list, then the expression also evaluates to `true`. If A is not null or empty, the expression evaluates to `false`.

Now that you've seen all the available operators, you can again update your precedence table to include these new operators:

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

You'll see much more about these operators in our discussion of using the EL with JavaBeans in the next section.

JavaBeans and the Expression Language

So far you've looked at the syntax of the EL. This in itself is not very useful when creating web applications. In this section, you'll focus on how to use the EL to read values from JavaBeans to display within a JSP page. In previous incarnations of the JSP specification, you would have had to use code such as the following to read values from a JavaBean:

```
<jsp:getProperty name="myBean" property="name" />
```

An alternative (and more common) method is to use a scriptlet such as the following:

```
<%= myBean.getName()%>
```

As we've discussed, the use of scriptlets does not represent good practice in JSP development. This may make you ask the question, "If I can use the `<getProperty>` standard action, why does anyone use scriptlets?" The answer to this question is simple: We developers are lazy! The scriptlet option represents less code and is a lot quicker to type!

To get around this problem, the EL provides a nice way to access the properties of a JavaBean that is in scope within a page, request, session, or application. To achieve the same as the previous code sample, you can use the following expression:

```
${myBean.name}
```

This is a nice neat way to access properties; there are no nasty brackets or any other Java-like syntax present. This brings us to another core feature of the EL: the concept of named variables. The EL provides a generalized mechanism for resolving variable names into objects. This mechanism has the same behavior as the `pageContext.findAttribute()` method of the `PageContext` object. Take the following, for example:

```
${product}
```

This expression will look for the attribute named `product` by searching the page, request, session, and application scopes, in that order, and will print its value. This works regardless of how the object was placed into the particular scope. That is, one JSP page could put an attribute into request, session, or application scope, and another JSP page could access the attribute simply by using an EL statement that uses the name of the attribute. If the attribute is not found, `null` will be returned. This method is also used to resolve the implicit objects that we'll talk about in the next section.

Listing 3-6 is a JSP page that uses EL to access JavaBeans.

Listing 3-6. *simpleBean.jsp*

```
<jsp:useBean id="person" class="com.apress.projsp.Person" scope="request">
  <jsp:setProperty name="person" property="*"/>
</jsp:useBean>
<html>
  <head>
    <title>EL and Simple JavaBeans</title>
    <style>
      body, td {font-family:verdana;font-size:10pt;}
    </style>
  <head>
  <body>
    <h2>EL and Simple JavaBeans</h2>
    <table border="1">
      <tr>
        <td>${person.name}</td>
        <td>${person.age}</td>
        <td>&nbsp;</td>
      </tr>
      <tr>
        <form action="simpleBean.jsp" method="post">
```

```
          <td><input type="text" name="name"></td>
          <td><input type="text" name="age"></td>
          <td><input type="submit"></td>
        </form>
      <tr>
    </table>
  </body>
</html>
```

The JSP page in Listing 3-6 uses a JavaBean of type `com.apress.projsp.Person`. That JavaBean is shown in Listing 3-7.

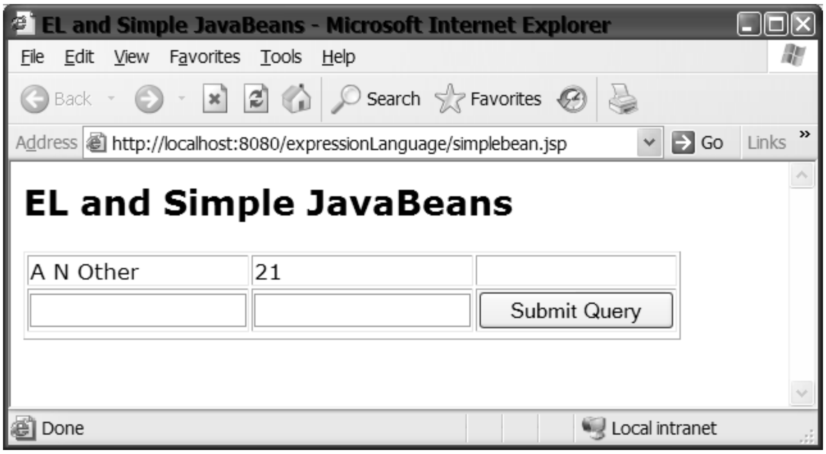
Listing 3-7. *Person.java*

```
package com.apress.projsp;
public class Person {
    private String name;
    private int age;
    public Person() {
        setName("A N Other");
        setAge(21);
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public int getAge() {
        return age;
    }
}
```

Use the following steps to deploy the JSP page and JavaBean into a web container:

1. Create a file called `simpleBean.jsp` in the `expressionLanguage` folder, and enter the code from Listing 3-6 into it.
2. Inside the `expressionLanguage\WEB-INF` directory, create a directory called `classes`.
3. Create a `com\apress\projsp` subdirectory within the `WEB-INF\classes` directory.
4. Create a file called `Person.java` within the `WEB-INF\classes\com\apress\projsp` directory with the contents shown in Listing 3-7, and compile it.
5. Start Tomcat and your web browser and go to `http://localhost:8080/expressionLanguage/simpleBean.jsp`.

You should see something similar to the page shown in Figure 3-6.


EL and Simple JavaBeans - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:8080/expressionLanguage/simplebean.jsp Go Links
Done Local intranet

EL and Simple JavaBeans

A N Other	21	
		Submit Query

Figure 3-6. *EL can be used to access JavaBeans and the properties of JavaBeans.*

The JSP page in Listing 3-6 creates a JavaBean of type `com.apress.projsp.Person` with an `id` of `person`, and sets its properties to the values of parameters in the HTTP request with the same name as the properties. This is achieved with the following code:

```
<jsp:useBean id="person" class="com.apress.projsp.Person" scope="request">
  <jsp:setProperty name="person" property="*"/>
</jsp:useBean>
```

The `<jsp:setProperty>` tag has various syntaxes. When you use `property="*"`, that tells the page implementation class to find each request parameter with the same name as a JavaBean property, and to set the JavaBean property with the value of the request parameter.

The JSP page accesses the object via the `id` given to it in the previous `<useBean>` tag, in this case, `person`. The page then displays the values of the properties of the `Person` JavaBean in a table; this is achieved by the following code:

```
<tr>
  <td>${person.name}<td>
  <td>${person.age}</td>
  <td>&nbsp;</td>
</tr>
```

The `id` is used to access the JavaBean that you declared with the previous `<useBean>` tag. In this example, the object was created and accessed within the same page. However, as we noted earlier, the object could have been created from any page, and still accessed in the `simpleBean.jsp` page.

It's worth noting that you could have used the following code to access the properties of our JavaBean:

```
<tr>
  <td>${person["name"]}</td>
  <td>${person["age"]}</td>
  <td>&nbsp;</td>
</tr>
```

Try changing the values in the form and clicking Submit Query. You should see your new values in the table. Now that you've seen a very simple use of JavaBeans and the EL, you'll look at a more complex use of the two technologies.

Nested Properties of a JavaBean

The EL provides you with a simple mechanism to access nested properties of a JavaBean. For example, Listing 3-8 shows a JavaBean, which has a nested property of type `Address` (Listing 3-9).

Listing 3-8. *Person.java*

```
package com.apress.projsp;

public class Person {
  private String name;
  private int age;
  private Address address;

  public Person() {
    setName("A N Other");
    setAge(21);

    this.address = new Address();
  }
  public void setName(String name) {
    this.name = name;
  }
  public String getName() {
    return name;
  }
  public void setAge(int age) {
    this.age = age;
  }
  public int getAge() {
    return age;
  }
  public void setAddress(Address address) {
    this.address = address;
  }
  public Address getAddress() {
    return address;
  }
}
```

As you can see, this JavaBean has a property that is in fact another JavaBean—the Address JavaBean (Address.java). Listing 3-9 shows the Address JavaBean.

Listing 3-9. *Address.java*

```
package com.apress.projsp;
import java.util.Collection;
public class Address {
  private String line1;
  private String town;
  private String county;
  private String postcode;
  private Collection phoneNumbers;
  public Address() {
    this.line1 = "line1";
    this.town = "a town2";
    this.county = "a county";
    this.postcode = "postcode";
  }
  public void setLine1(String line1) {
    this.line1 = line1;
  }

  public String getLine1() {
    return line1;
  }
  public void setTown(String town) {
    this.town = town;
  }
  public String getTown() {
    return town;
  }
  public void setCounty(String county) {
    this.county = county;
  }
  public String getCounty() {
    return county;
  }
  public void setPostcode(String postcode) {
    this.postcode = postcode;
  }
  public String getPostcode() {
    return postcode;
  }
  public Collection getPhoneNumbers() {
    return phoneNumbers;
  }
  public void setPhoneNumbers(Collection phoneNumbers) {
    this.phoneNumbers = phoneNumbers;
  }
}
```

It's simple to access these nested properties by using the EL. With the EL, you can chain the various objects and properties in an EL statement. The following JSP snippet shows how you could use chaining to access the `line1` property of the `address` property of the `person` JavaBean.

```
${person.address.line1}
```

The `Address` JavaBean contains a collection of other JavaBeans as one of its properties. Although you can't see it in Listing 3-9, this collection is a collection of JavaBeans—`PhoneNumber` JavaBeans. This JavaBean is shown in Listing 3-10.

Listing 3-10. *PhoneNumber.java*

```
package com.apress.projsp;
public class PhoneNumber {
  private String std;
  private String number;

  public String getNumber() {
    return number;
  }
  public String getStd() {
    return std;
  }
  public void setNumber(String number) {
    this.number = number;
  }
  public void setStd(String std) {
    this.std = std;
  }
}
```

The EL also provides a simple mechanism for accessing such collections and the properties of their enclosed JavaBeans. The following JSP snippet would access the first phone number for a person's address:

```
${person.address.phoneNumbers[1].number}
```

As this snippet shows, you can freely mix both dot and bracket notation as needed to access JavaBean properties.

We can bring this whole discussion together by way of the following example. Listing 3-11 shows a JSP page that displays all the details relating to a person and that person's address. Note how the JSP page uses alternative syntaxes for object property access.

Listing 3-11. *complexBean.jsp*

```
<html>
<head>
  <title>EL and Complex JavaBeans</title>
  <style>
    body, td {font-family:verdana;font-size:10pt;}
  </style>
</head>
<body>
  <h2>EL and Complex JavaBeans</h2>
  <table border="1">
    <tr>
      <td>${person.name}</td>
      <td>${person.age}</td>
      <td>${person["address"].line1}</td>
      <td>${person["address"].town}</td>
      <td>${person.address.phoneNumbers[0].std}
          ${person.address.phoneNumbers[0].number}</td>
      <td>${person.address.phoneNumbers[1].std}
          ${person.address.phoneNumbers[1].number}</td>
    </tr>
  </table>
</body>
</html>
```

Unlike previous examples where you loaded the JSP page directly, this example uses a simple Java servlet (Listing 3-12) to set up the information within the JavaBeans. The servlet then adds the object to the request by using the name "person." The JSP page, as mentioned earlier, searches the various scopes to find the JavaBean with the name used in the EL statement.

Listing 3-12, `PopulateServlet.java`, is the servlet that initializes the various JavaBeans and then uses a `RequestDispatcher` to forward the request to `complexBean.jsp`. To compile this class, you'll need to include a servlet library in the CLASSPATH. If you are using Tomcat 5, you can find the library, `servlet-api.jar`, in the `common\lib` directory. If you are using some other web container, check your container documentation for the correct servlet library to include.

Listing 3-12. *PopulateServlet.java*

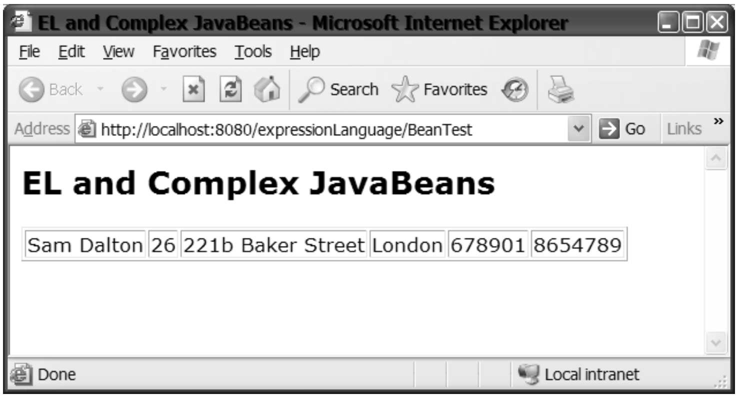
```
package com.apress.projsp;
import java.io.IOException;
import java.util.ArrayList;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class PopulateServlet extends HttpServlet {
  protected void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
  {

    Person p = new Person();
    p.setName("Sam Dalton");
    p.setAge(26);
    Address a = new Address();
    a.setLine1("221b Baker Street");
    a.setTown("London");
    a.setCounty("Greater London");
    a.setPostcode("NW1 1AA");
    ArrayList al = new ArrayList();
    PhoneNumber ph = new PhoneNumber();
    ph.setStd("01895");
    ph.setStd("678901");
    al.add(ph);

    ph = new PhoneNumber();
    ph.setStd("0208");
    ph.setStd("8654789");
    al.add(ph);
    a.setPhoneNumbers(al);
    p.setAddress(a);
    req.setAttribute("person", p);
    RequestDispatcher rd = req.getRequestDispatcher("complexBean.jsp");
    rd.forward(req, res);
  }
}
```

This servlet class should be placed within the `WEB-INF\classes\com\apress\projsp` folder. You'll also need to modify the `WEB-INF\web.xml` file to contain the additional tags in Listing 3-13.

Listing 3-13. *web.xml*

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation= ➥
"http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
         version="2.5">

  <servlet>
        <servlet-name>BeanTestServlet</servlet-name>
        <servlet-class>com.apress.projsp.PopulateServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>BeanTestServlet</servlet-name>
        <url-pattern>/BeanTest</url-pattern>
    </servlet-mapping>
</web-app>
```

Compile all the Java classes and deploy the files to a web container in the manner described for previous examples. The request for this application is `http://localhost:8080/expressionLanguage/BeanTest`. The request is passed to the `BeanTestServlet`. The servlet creates and initializes the various JavaBeans and forwards the request to `complexBean.jsp`. Figure 3-7 shows the web page you would see in your browser.


EL and Complex JavaBeans - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:8080/expressionLanguage/BeanTest Go Links
Done
Local intranet

EL and Complex JavaBeans

Sam Dalton	26	221b Baker Street	London	678901	8654789

Figure 3-7. *The expression language can easily handle nested properties of JavaBeans.*

This example shows how an object or attribute can be created in one part of the web application and accessed from some other part of the application. The JSP page searches the various scopes until it locates the named object.

Expression-Language Implicit Objects

Within JSP scriptlets you have many implicit objects available to you. These objects allow you to access things such as the request, session, and page context. The EL also provides you with these implicit objects, and a lot more besides. The objects are always available under well-known names and are resolved to objects in the same way as JavaBeans. Table 3-1 lists the implicit objects, along with brief descriptions of each object.

Table 3-1. *JSP Implicit Objects*

Implicit Object	Description
`applicationScope`	This is a `Map` that contains all application-scoped variables. The `Map` is keyed on the name of the variable.
`cookie`	This is a `Map` that maps cookie names to a single `Cookie` object. If more than one cookie exists for a given name, the first of these cookies is used for that name.
`header`	This is a `Map` that contains the values of each header name.

Implicit Object	Description
headerValues	This is a Map that maps a header name to a string array of all the possible values for that header.
initParam	This is a Map that maps context initialization parameter names to their string parameter values.
pageContext	The PageContext object.
pageScope	This is a Map that contains all page-scoped variables. The Map is keyed on the name of the variable.
param	This is a Map that contains the names of the parameters to a page. Each parameter name is mapped to a single string value.
paramValues	This is a Map that maps a parameter name to a string array of all the values for that parameter.
requestScope	This is a Map that contains all request-scoped variables. The Map is keyed on the name of the variable.
sessionScope	This is a Map that contains all session-scoped variables. The Map is keyed on the name of the variable.

Listing 3-14 shows an example JSP page that uses some of these implicit objects.

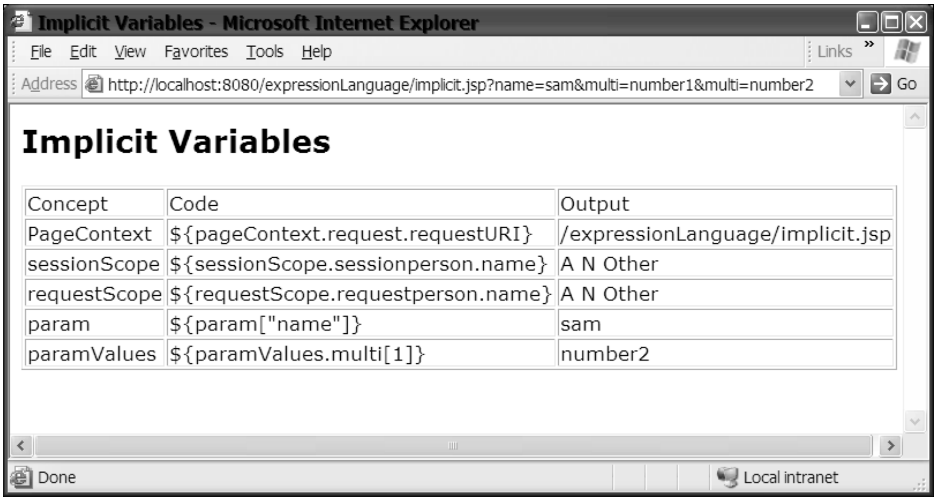
Listing 3-14. *implicit.jsp*

```
<jsp:useBean id="sessionperson" class="com.apress.projsp.Person"
             scope="session" />
<jsp:useBean id="requestperson" class="com.apress.projsp.Person"
             scope="request" />
<html>
  <head>
    <title>Implicit Variables</title>
    <style>
      body, td {font-family:verdana;font-size:10pt;}
     </style>
  </head>
  <body>
    <h2>Implicit Variables</h2>
    <table>
      <tr>
        <td>Concept</td>
        <td>Code</td>
        <td>Output</td>
      </tr>
      <tr>
        <td>PageContext</td>
        <td>${'${'}pageContext.request.requestURI}</td>
        <td>${pageContext.request.requestURI}</td>
      </tr>
      <tr>
        <td>sessionScope</td>
```

```
          <td>${'${'}sessionScope.sessionperson.name}</td>
          <td>${sessionScope.sessionperson.name}</td>
        </tr>
        <tr>
          <td>requestScope</td>
          <td>${'${'}requestScope.requestperson.name}</td>
          <td>${requestScope.requestperson.name}</td>
        </tr>
        <tr>
          <td>param</td>
          <td>${'${'}param["name"]}</td>
          <td>${param["name"]}</td>
        </tr>
        <tr>
          <td>paramValues</td>
          <td>${'${'}paramValues.multi[1]}</td>
          <td>${paramValues.multi[1]}</td>
        </tr>
      </table>
    </body>
</html>
```

This example shows how to use the request- and session-scope maps, the request parameter map, and the request parameter values map as well as the `pageContext` object. All the other objects are used in exactly the same manner and are not shown here.

If you deploy this example to a web container (as described in the previous sections) and request the page with a URL similar to `http://localhost:8080/expressionLanguage/implicit.jsp?name=sam&multi=number1&multi=number2`, you'll see a page similar to that shown in Figure 3-8.


Implicit Variables - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Links
Address
Go
Done
Local intranet

http://localhost:8080/expressionLanguage/implicit.jsp?name=sam&multi=number1&multi=number2

Implicit Variables

Concept	Code	Output
PageContext	${pageContext.request.requestURI}	/expressionLanguage/implicit.jsp
sessionScope	${sessionScope.sessionperson.name}	A N Other
requestScope	${requestScope.requestperson.name}	A N Other
param	${param["name"]}	sam
paramValues	${paramValues.multi[1]}	number2

Figure 3-8. *Using implicit variables, you can access various objects that provide information about the application, the page, the request, the session, and so on.*

Expression-Language Functions

This final section discusses perhaps the most interesting part of the JSP EL: functions. An EL function is mapped to a static method of a Java class. This mapping is specified within a tag library descriptor (TLD), as you'll see later.

As with the rest of the EL, a function can appear either in template text or in the attributes of a custom tag.

A function in EL can take any number of parameters, and these are again declared in a deployment descriptor. Functions are assigned a namespace that is used to access a function similar to package specifications in Java classes; for example, the following JSP code invokes an EL function:

```
${MyFunctions:function("param")}
```

The namespace in this example is `MyFunctions`, which is declared by using this `taglib` directive:

```
<%@ taglib uri="/WEB-IF/taglib.tld" prefix="MyFunctions" %>
```

You'll see this directive used in other places in this book to declare namespaces for custom tags; in this chapter you'll use it to declare namespaces for EL functions. Functions must always have a namespace, with one exception: if a function is used within the attribute of a custom tag, the function call may omit the namespace as long as the function is declared within the same TLD as the custom tag.

A TLD is an XML file that declares a tag library. The TLD contains information relating to the tags in the library and the classes that implement them. The TLD also contains the declarations and mappings of EL functions. Each TLD can describe zero or more static functions. Each function is given a name and a specific method in a Java class that will implement the function. The method must be a public static method on a public class. If this is not the case, translation-time error will occur. Within a TLD, function names must be unique, and if two functions have the same name, a translation-time error will occur. Here is an example of the TLD entries used to declare a function:

```
<taglib>
...
  <function>
    <name>nickname</name>
    <function-class>com.apress.projsp.Functions</function-class>
    <function-signature>
      java.lang.String nickname(java.lang.String)
    </function-signature>
  </function>
</taglib>
```

This TLD fragment declares a function called `nickname`, which is intended to return the nickname for a particular user. If you look at the tags, you can see that we declared the name of the function that will be used by the EL by using the `<name>` element, the class that implements the function by using the `<function-class>` element, and the signature of the function by using the `<function-signature>` element. It's this last element that is the most interesting and also the most complex.

The syntax of the `<function-signature>` element is as follows:

```
return_type static_function_name(parameter_1_type,..,parameter_n_type)
```

It's important to note that the parameter and return types must be the fully qualified Java class names. If the declared signature of the function does not match that of the function in the Java class, a translation-time error will occur.

For more information about the content of TLDs, especially those that relate to custom tags, see Chapters 6 through 8.

A Simple Function

You are now ready to look at some example functions. The first of these examples outputs a greeting to the user. This greeting is customized depending on the time of day. So, for example, if called before midday the greeting will be `Good Morning`; if called after midday but before 6 p.m., the greeting will be `Good Afternoon`; and finally, if called after 6 p.m. but before midnight, the greeting will be `Good Evening`. Listing 3-15, `Functions.java`, is the Java class that implements this function.

Listing 3-15. *Functions.java*

```
package com.apress.projsp;
import java.util.Calendar;
public class Functions {
  public static String sayHello() {
    Calendar rightNow = Calendar.getInstance();
    int hour = rightNow.get(Calendar.HOUR);
    int AmPm = rightNow.get(Calendar.AM_PM);

    if (AmPm == Calendar.AM) {
      return "Good Morning";
    } else if (AmPm == Calendar.PM && hour < 6) {
      return "Good Afternoon";
    } else {
      return "Good Evening";
    }
  }
}
```

The TLD for this function is very simple, because the function has no parameters. Listing 3-16 shows the TLD, which is named `taglib.tld` and is located in the `tags` folder under the `WEB-INF` folder.

Listing 3-16. *taglib.tld*

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib xmlns="http://java.sun.com/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation= ➥
```

```
"http://java.sun.com/xml/ns/javaee/web-jsptaglibrary_2_1.xsd"
        version="2.1">

  <tlib-version>1.0</tlib-version>
  <jsp-version>2.1</jsp-version>
  <short-name>projsp</short-name>
  <uri>/projsp</uri>
  <description>
  </description>
  <function>
    <name>greet</name>
    <function-class>com.apress.projsp.Functions</function-class>
    <function-signature>java.lang.String sayHello()</function-signature>
  </function>
</taglib>
```

This TLD declares our function to have the name `greet` and to return a string and take no parameters. You can use this function in a JSP page, as shown in Listing 3-17. Notice how you declare the namespace for your tag library as `projsp` in the first line, and that the function name is prefixed with `projsp`. The function is called by using the name provided by the TLD, rather than by the actual function name in the Java class file.

Listing 3-17. *greet.jsp*

```
<%@ taglib prefix="projsp" uri="/WEB-INF/tags/taglib.tld"%>
<html>
  <head>
    <title>Greet</title>
  </head>
  <body>
    <pre>${projsp:greet()}</pre>
  </body>
</html>
```

With this example, we need to make an addition to the deployment descriptor. Listing 3-18 shows the addition to make:

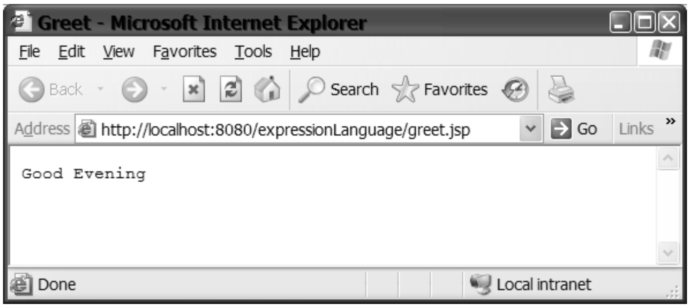
Listing 3-18. *tag-lib (Addition to web.xml)*

```
    <taglib>
        <taglib-uri>
           /chapter3
        </taglib-uri>
        <taglib-location>
           /WEB-INF/tags/taglib.tld
        </taglib-location>
    </taglib>
```

To deploy this example, perform the following steps:

1. Add the code shown in Listing 3-18 to `web.xml`.
2. Create and compile the source code for `Functions.java`, shown in Listing 3-15, and save it to the `WEB-INF\classes\com\apress\projsp` directory.
3. Create the JSP page `greet.jsp` (Listing 3-17) and save it to the `expressionLanguage` folder.
4. Create a folder within the `WEB-INF` directory called `tags`, and a file called `taglib.tld` within this folder. The contents of `taglib.tld` are shown in Listing 3-16.
5. Start Tomcat, open your web browser, and go to `http://localhost:8080/expressionLanguage/greet.jsp`.

You should see a page similar to that shown in Figure 3-9. As expected, the output of this JSP page when viewed after 6 p.m but before midnight contains the greeting “Good Evening.” If you view the page at different times of the day, you will see different greetings.


Greet - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:8080/expressionLanguage/greet.jsp
Go Links
Done
Local intranet

```
Good Evening
```

Figure 3-9. *The EL can be used to call static functions of Java classes in the web application.*

A More Complex Function

Having looked at a very simple function that takes no parameters, you are now in a position to look at a more complex function. This function allows you to view the untranslated source of a JSP page presented in HTML format. To do this, the function will accept two parameters. Add the method shown in Listing 3-19 to the `Functions.java` source file that was first presented in Listing 3-15.

Listing 3-19. *source() (Addition to Functions.java)*

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import javax.servlet.jsp.PageContext;
//...
  public static String source(String filename, PageContext pageContext)
    throws IOException {
```

```
    // use the servlet context to read in the file
    InputStream in;
    BufferedReader br;
    StringBuffer buf = new StringBuffer();

    in = pageContext.getServletContext().getResourceAsStream(filename);
    br = new BufferedReader(new InputStreamReader(in));
    String line = br.readLine();
    while (line != null) {
      // replace opening and closing tags
      line = line.replaceAll("<", "&lt;");
      line = line.replaceAll(">", "&gt;");
      // writing out each line as we go
      buf.append(line + "\n");
      line = br.readLine();
    }
    br.close();
    // return the contents of the file
    return buf.toString();
  }
//...
```

You'll notice that this function uses `pageContext` to read in the JSP file to be displayed. In addition to importing the package for `pageContext`, you will need to include the `jsp-api.jar` library in the CLASSPATH to compile the class. For Tomcat 5, the `jsp-api.jar` library is in the `common\lib` directory. If you have some other container, consult your container's documentation to determine the correct library. The `pageContext` is passed in as a parameter, and as such it must be declared in the function signature. The `<function>` element for this function in the TLD is shown in Listing 3-20. You need to add this element to the TLD in Listing 3-16.

Listing 3-20. *<function> (Addition to taglib.tld)*

```
  <function>
    <name>source</name>
    <function-class>com.apress.projsp.Functions</function-class>
    <function-signature>
      java.lang.String source(java.lang.String, javax.servlet.jsp.PageContext)
    </function-signature>
  </function>
```

As you can see, we've declared the second parameter to this function to be of type `javax.servlet.jsp.PageContext`.

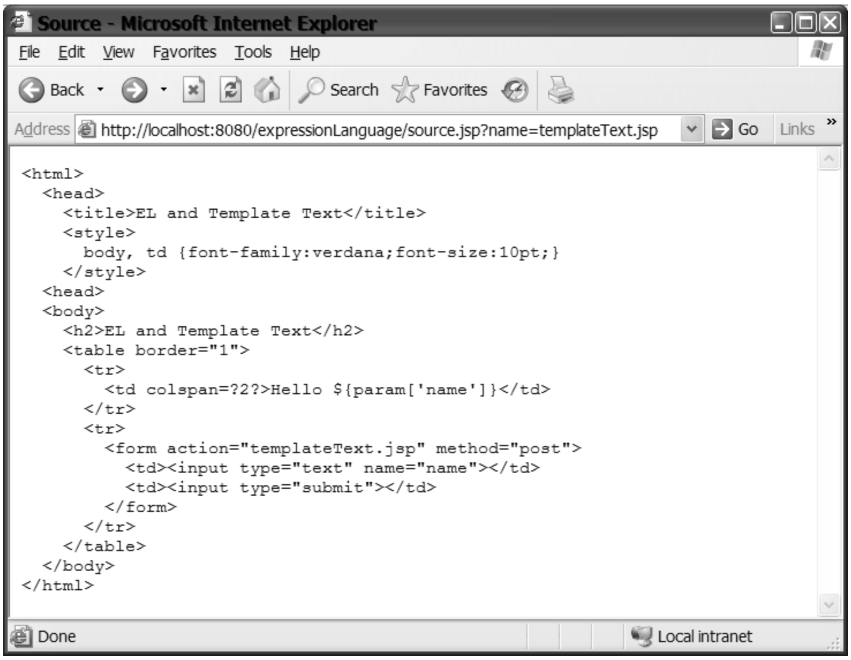
Listing 3-21 shows a JSP page that calls the function. Notice that that `taglib` directive defines the prefix for the tag to be `"Source"`. This is in contrast to the `<short-name>` element of the TLD (Listing 3-16). The JSP specification says that the `<short-name>` element is the preferred prefix for tags defined in the TLD. However, the `prefix` attribute in the `taglib` directive overrides the TLD. The function is thus called with the prefix (from the `taglib` directive) and name (from the TLD). Two arguments are passed to the function: the request parameter named `name` and the `pageContext` implicit object.

Listing 3-21. *source.jsp*

```
<%@ taglib prefix="Source" uri="/WEB-INF/tags/taglib.tld"%>
<html>
  <head>
    <title>Source</title>
  </head>
  <body>
    <pre>${Source:source(param.name, pageContext)}</pre>
  </body>
</html>
```

Add Listing 3-19 to `Functions.java` and compile that file. Also, add Listing 3-20 to `taglib.tld`. If Tomcat is running, you will need to restart Tomcat, or reload the `expressionLanguage` web application so that Tomcat will reload the class file.

Open a web browser and enter `http://localhost:8080/expressionLanguage/source.jsp?templateText.jsp` into the address bar. If you created `templateText.jsp` from Listing 3-1, Figure 3-10 shows an example of the page you might see. Notice that the name of the file to be displayed is passed as a request parameter in the URL. You can use this JSP page to view the source of any resource in the `expressionLanguage` web application, as long as you pass the correct path to the file. You cannot use `source.jsp` to view sources in other web applications, because this page has access only to the files that can be seen in the current `pageContext`.


Source - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:8080/expressionLanguage/source.jsp?name=templateText.jsp
Go Links
Done
Local intranet

```
<html>
  <head>
    <title>EL and Template Text</title>
    <style>
      body, td {font-family:verdana;font-size:10pt;}
    </style>
  <head>
  <body>
    <h2>EL and Template Text</h2>
    <table border="1">
      <tr>
        <td colspan=?2?>Hello ${param['name']}</td>
      </tr>
      <tr>
        <form action="templateText.jsp" method="post">
          <td><input type="text" name="name"></td>
          <td><input type="submit"></td>
        </form>
      </tr>
    </table>
  </body>
</html>
```

Figure 3-10. *You can pass any objects available in the JSP page to an EL function. In this example, a String and a PageContext object were passed to a function.*

Functions in Tag Attributes

One of the most powerful uses of functions in the EL is to preprocess the attributes passed to standard and custom tags. In this section, you'll look at such a use of the EL. You'll write a function that will convert a tag's parameter to uppercase; although this is not a particularly complex function, it should suffice. We don't want to get bogged down in the details of the function's implementation.

Let's first take a look at the Java method that provides this function, shown in Listing 3-22.

Listing 3-22. *toUpperCase() (Addition to Functions.java)*

```
  public static String toUpperCase(String theString) {
    return theString.toUpperCase();
  }
```

This is an extremely simple function that merely converts its parameter to uppercase. The TLD entry for the function is shown in Listing 3-23.

Listing 3-23. *<function> (Addition to taglibs.tld)*

```
  <function>
    <name>upper</name>
    <function-class>com.apress.projsp.Functions</function-class>
    <function-signature>
      java.lang.String toUpperCase(java.lang.String)
    </function-signature>
  </function>
```

In this example, you'll create a JavaBean and use the standard tag `<jsp:setProperty>` to set the JavaBean's property. Listing 3-24 shows this simple JavaBean, `SourceBean.java`. To show the use of functions with tag attributes, we'll use a function to preprocess the attribute value.

Listing 3-24. *SourceBean.java*

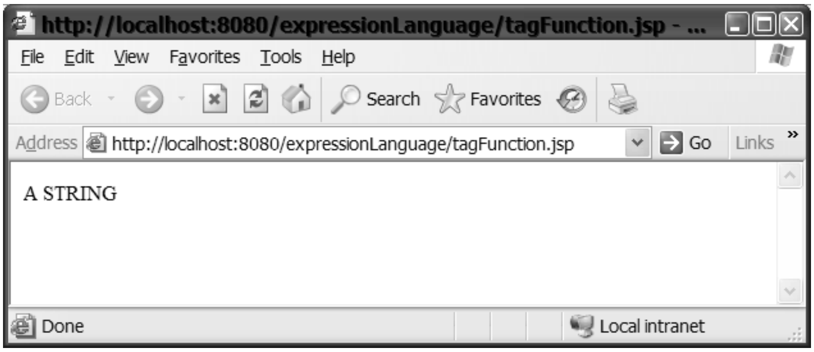
```
package com.apress.projsp;

public class SourceBean {
  String string;
  public String getString() {
    return string;
  }
  public void setString(String s) {
    string = s;
  }
}
```

Now that you've seen the constituent parts, you'll look at a JSP page in Listing 3-25 that uses a function to preprocess the attribute to the `<jsp:setProperty>` tag.

Listing 3-25. *tagFunction.jsp*

```
<%@ taglib prefix="projsp" uri="/WEB-INF/tags/taglib.tld"%>
<html>
  <body>
    <jsp:useBean id="sb" class="com.apress.projsp.SourceBean"/>
    <jsp:setProperty name="sb" property="string"
                     value="${projsp:upper('a string')}" />
    ${sb.string}
  </body>
</html>
```

This JSP page creates a JavaBean, uses a `<jsp:setProperty>` tag to set its property, and then displays the value of the property by using the EL statement `${sb.string}`. In the `<jsp:setProperty>` tag, you can see that we use a function to preprocess the string passed to the value attribute. Functions can be used to preprocess the attributes of all standard tags and custom tags (which you will see in later chapters). To deploy this to a web container, follow the steps for the other examples in this chapter. Figure 3-11 shows the web page generated by the JSP page.


http://localhost:8080/expressionLanguage/tagFunction.jsp - ...
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:8080/expressionLanguage/tagFunction.jsp
Go Links
Done
Local intranet

A STRING

Figure 3-11. *The value of the attribute was changed to uppercase before the property of the JavaBean was set. When the property is displayed in a web page, the property is displayed in its processed form.*

We've deliberately made this example simplistic, so that we can focus on the concept of calling a function in a tag attribute by using EL. However, you can use this technique to perform any processing that might be needed.

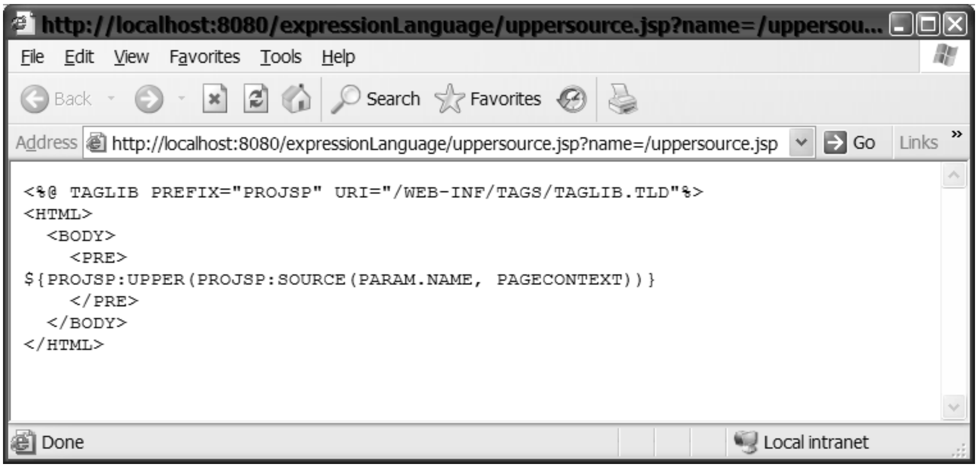
Nesting Functions

Another powerful use of functions is to nest them together. For example, you can use the uppercase function from the previous example (Listings 3-22 and 3-25) to render the source of a page produced by our view-source function in uppercase. Listing 3-26, `uppersource.jsp`, shows how you might do this.

Listing 3-26. *uppersource.jsp*

```
<%@ taglib prefix="projsp" uri="/WEB-INF/tags/taglib.tld"%>
<html>
  <body>
    <pre>
${projsp:upper(projsp:source(param.name, pageContext))}
    </pre>
  </body>
</html>
```

Deploy this JSP page, and then access it with the URL `http://localhost:8080/expressionLanguage/uppersource.jsp?name=uppersource.jsp`. Figure 3-12 shows the page that is generated.


http://localhost:8080/expressionLanguage/uppersource.jsp?name=/uppersou...
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:8080/expressionLanguage/uppersource.jsp?name=/uppersource.jsp Go Links
Done Local intranet

```
<%@ TAGLIB PREFIX="PROJSP" URI="/WEB-INF/TAGS/TAGLIB.TLD"%>
<HTML>
  <BODY>
    <PRE>
${PROJSP:UPPER(PROJSP:SOURCE(PARAM.NAME, PAGECONTEXT))}
    </PRE>
  </BODY>
</HTML>
```

Figure 3-12. *EL functions can be nested. This example uses the source() function to access a source file, and then the upper() function to convert the source to uppercase.*

Functions can be nested to an arbitrary degree to perform some interesting and powerful operations. The only restricting factor is your imagination. This nesting encourages small functions that perform specialized jobs, which is a very positive design point.

Expression-Language Functions vs. Custom Tags

As you'll see in later chapters, the JSP specification provides a powerful custom tag mechanism. You might ask why you would use tags over functions and vice versa. There are several factors that can help you to make the choice:

- Is knowledge of the environment required? If the answer is yes, tags are the way to go. A tag provides easy access to `pageContext` and other variables; functions do not. To access these implicit objects within a function, you must pass them in as a parameter.

- Do you require iterative behavior over a body? If the answer is yes, you should use a tag. Functions do not provide functionality to process a body (they don't have one), whereas tags do.
- Are you trying to provide a small, reusable piece of functionality that acts on one or more arguments? If the answer to this is yes, you should use a function. Overall, functions are much simpler to write than tags; therefore, they provide a great opportunity to write small, self-contained pieces of functionality.
- Would you like to reuse existing Java code in a web context? If the answer is yes, functions are ideal. Because functions are no more than static Java methods, you can easily reuse existing code.

The choice of tags versus functions should be eased by consulting these points, but it's worth noting that the true power of the EL becomes evident when it's combined with custom tags.

Summary

In this chapter, you've looked at the JSP EL. This EL is largely intended to replace scriptlets and to be used in combination with custom tags.

You've examined the following topics in this chapter:

- The reasons that the EL has come about, including a look at its history
- The syntax and usage of the EL, including reserved words, disabling scriptlets in a page, and disabling the evaluation of the EL on a page or set of pages
- The operators within the EL, including arithmetic operators, comparison operators, logical operators, and other operators
- The use of JavaBeans with the EL
- The implicit objects within the EL
- The declaration and use of functions in the EL, including reasons for using functions over tags and vice versa

In the next chapter, you'll learn about the JSTL and the tags contained within it.