8

How to use EL

In chapter 6, you learned how to code a JavaBean. In addition, you learned how to use the Expression Language (EL) that was introduced with JSP 2.0 to get properties from that JavaBean. Now, you'll learn more about using EL. In addition, you'll see that EL is a significant improvement over the standard JSP tags that were typically used prior to JSP 2.0.

If you skipped chapter 6, you should still be able to understand this chapter, as long as you know how to code a JavaBean. If you don't, you should go back to chapter 6 and read about JavaBeans at the start of that chapter. Then, you'll be ready for this chapter.

An introduction to JSP Expression Language

The JSP *Expression Language* (*EL*) provides a compact syntax that lets you get data from JavaBeans, maps, arrays, and lists that have been stored as attributes of a web application. To illustrate, figure 8-1 presents two examples that get data from a User object named user that has been stored as an attribute of the session object.

Both of these examples assume that the User class follows all the rules for creating a JavaBean that are shown in figure 6-1 of chapter 6. Then, the first example uses EL to get the properties of the User bean, and the second example uses standard JSP tags to get those properties. This should give you a quick idea of how EL can improve your JSPs.

Advantages of EL

EL has several advantages over standard JSP tags. First, EL is more compact and elegant. This makes it easier to code and read.

Second, although it isn't shown in this figure, EL makes it easy to access nested properties. For example, you can access a property named code from a JavaBean named product that's in a JavaBean named item like this:

```
${item.product.code}
```

Third, although standard JSP tags only let you access JavaBeans, EL lets you access collections such as arrays, maps, and lists. A *map* is a collection that implements the Map interface, such as a HashMap collection. A *list* is a collection that implements the List interface, such as the ArrayList<> collection. For more information about arrays and collections, you can refer to *Murach's Java Programming*.

Fourth, EL usually handles null values better than standard JSP tags. For example, instead of returning a null value for a string variable, EL returns an empty string, which is usually what you want.

Fifth, EL provides functionality that isn't available from the standard JSP tags. For example, it lets you work with HTTP headers, cookies, and context initialization parameters. It also lets you perform calculations and comparisons.

Disadvantages of EL

Unlike the standard JSP tags, EL doesn't create a JavaBean if the JavaBean hasn't already been stored as an attribute. In addition, EL doesn't provide a way to set the properties of a JavaBean. However, when you use the MVC pattern, you typically use a servlet to create a JavaBean, set its properties, and store it as an attribute. As a result, these disadvantages aren't an issue. In fact, many developers would say that having the view directly create JavaBeans and set their properties, violates good MVC practices.

Code in a JSP that accesses a User object that's stored in the session object

EL

```
<label>Email:</label>
<span>${user.email}</span><br>
<label>First Name:</label>
<span>${user.firstName}</span><br>
<label>Last Name:</label>
<span>${user.lastName}</span><br>
```

Standard JSP tags

```
<jsp:useBean id="user" scope="session" class="murach.business.User"/>
<label>Email:</label>
<span><jsp:getProperty name="user" property="email"/></span><br>
<label>First Name:</label>
<span><jsp:getProperty name="user" property="firstName"/></span><br>
<label>Last Name:</label>
<span><jsp:getProperty name="user" property="lastName"/></span><br>
```

Advantages of EL

- EL has a more elegant and compact syntax than standard JSP tags.
- EL lets you access nested properties.
- EL lets you access collections such as maps, arrays, and lists.
- EL does a better job of handling null values.
- EL provides more functionality.

Disadvantages of EL

- EL doesn't create a JavaBean if it doesn't already exist.
- EL doesn't provide a way to set properties.

Description

- The JSP *Expression Language* (*EL*) provides a compact syntax that lets you get data from JavaBeans, maps, arrays, and lists that have been stored as attributes of a web application.

Figure 8-1 An introduction to EL

Essential skills for working with EL

Now that you have a general idea of how EL can simplify and improve your JSP code, you're ready to learn the details of working with it.

How to use the dot operator to work with JavaBeans and maps

As you learned in chapters 5 and 7, you can use the setAttribute method of the HttpServletRequest and HttpSession objects to store an object with request or session scope. If you need a larger scope, you can use the setAttribute method of the ServletContext object to store an object with application scope. Or, if you need a smaller scope, you can use the setAttribute method of the implicit PageContext object to store an object with page scope. Then, you can use the getAttribute method of the appropriate object to retrieve the attribute.

Figure 8-2 shows how to use EL to access an attribute of a web application. Whenever you use EL, you begin by coding a dollar sign ($) followed by an opening brace ({) and a closing brace (}). Then, you code the expression within the braces.

The first example in this figure shows how to retrieve an attribute for a simple object like a String or Date object. Here, the servlet code creates a Date object named currentDate that stores the current date. Then, the servlet code stores this object as an attribute of the request object. Last, the JSP code uses EL to access this attribute, convert it to a string, and display it.

Note here that you don't have to specify the scope when you use EL. Instead, EL automatically searches through all scopes starting with the smallest scope (page scope) and moving towards the largest scope (application scope).

The second example shows how to display a property of an attribute for a more complex object like a JavaBean or a map. Here, the servlet code creates a JavaBean for the user and stores this bean as an attribute of the session. Then, the JSP code uses EL to access this attribute, and it uses the dot operator to specify the property of the JavaBean that it's going to display.

You can use the same technique to work with a map. In that case, though, you code the name of the key after the dot operator to get the associated object that's stored in the map. You'll see this used in the next figure.

An example that accesses the currentDate attribute

Syntax

```
${attribute}
```

Servlet code

```
Date currentDate = new Date();
request.setAttribute("currentDate", currentDate);
```

JSP code

```
<p>The current date is ${currentDate}</p>
```

An example that accesses the firstName property of the user attribute

Syntax

```
${attribute.property}
```

Servlet code

```
User user = new User(firstName, lastName, email);
session.setAttribute("user", user);
```

JSP code

```
<p>Hello ${user.firstName}</p>
```

The sequence of scopes that Java searches to find the attribute

Scope	Description
`page`	The bean is stored in the implicit PageContext object.
`request`	The bean is stored in the HttpServletRequest object.
`session`	The bean is stored in the HttpSession object.
`application`	The bean is stored in the ServletContext object.

Description

- A JavaBean is a special type of object that provides a standard way to access its *properties*.
- A *map* is a special type of collection that's used to store key/value pairs. For example, a HashMap collection is a map.
- When you use the dot operator, the code to the left of the operator must specify a JavaBean or a map, and the code to the right of the operator must specify a JavaBean property or a map key.
- When you use this syntax, EL looks up the attribute starting with the smallest scope (page scope) and moving towards the largest scope (application scope).

Figure 8-2 How to use the dot operator to work with JavaBeans and maps

How to use EL to specify scope

Since Java automatically searches through the scope objects when you use EL, you typically don't need to use the implicit EL objects shown in figure 8-3 for specifying scope. However, if you have a naming conflict, you may need to use them. When you work with these objects, you should be aware that they are all maps. As a result, you can use the dot operator to specify a key when you want to return the object for that key.

To illustrate the use of these EL objects, this figure presents the same examples as in figure 8-3. However, they work even if there is an attribute with the same name stored in a larger scope. Here, the first example uses request scope to identify the user object. The second example uses session scope.

The implicit EL objects for specifying scope

Scope	Implicit EL object
page	pageScope
request	requestScope
session	sessionScope
application	applicationScope

An example that specifies request scope

Syntax

```
${scope.attribute}
```

Servlet code

```
Date currentDate = new Date();
request.setAttribute("currentDate", currentDate);
```

JSP code

```
<p>The current date is ${requestScope.currentDate}</p>
```

An example that specifies session scope

Syntax

```
${scope.attribute.property}
```

Servlet code

```
User user = new User(firstName, lastName, email);
session.setAttribute("user", user);
```

JSP code

```
<p>Hello ${sessionScope.user.firstName}</p>
```

Description

- If you have a naming conflict, you can use the *implicit EL objects* to specify scope.
- All of the implicit EL objects for specifying scope are maps. As a result, you can use the dot operator to specify a key when you want to return the object for that key.

Figure 8-3 How to use EL to specify scope

How to use the [ ] operator to work with arrays and lists

Figure 8-4 shows how to use the [ ] operator. Although this operator can be used to work with JavaBeans and maps, it is commonly used to work with arrays and lists.

The first example in this figure shows how to use the [ ] operator to access the firstName property of an attribute named user. This has the same effect as the second example in figure 8-2. However, the second example in that figure is easier to code and read. That's why you'll typically use the dot operator to access the properties of JavaBeans and the values of maps.

The primary exception to this is if you use a dot in a name. If, for example, you use a map key named "murach.address", you can't use the dot operator to access that value. However, you can use the [ ] operator. If necessary, you can use an implicit EL object to specify the scope so you can use the [ ] operator to access an attribute that uses a dot in its name.

The second example shows how to use the [ ] operator to access an array of strings. Here, the servlet code creates an array of strings that stores three colors. Then, it gets a ServletContext object so it can store the array in this object, which makes it available to the entire application. Finally, the JSP code uses EL to retrieve the first two strings that are stored in the array.

Note that this code is similar to the Java syntax for accessing strings that are stored in an array, and that the index values can be enclosed in quotation marks. Although quotation marks are required for using the [ ] operator to access a property of a JavaBean or a key of a map, they are optional for specifying the index of an array or a list.

The third example shows how to use the [ ] operator to access a list of User objects. Here, the servlet code uses the getUsers method of the UserIO class to retrieve a list of User objects. Then, the servlet code stores the list in the session object. Finally, the JSP code uses the [ ] operator to access the first two User objects, and it uses the dot operator to display the emailAddress property of these objects. This shows that you can mix the [ ] and dot operators if that's required.

The syntax for the [ ] operator

```
${attribute["propertyKeyOrIndex"]}
```

An example that works with a JavaBean property

Servlet code

```
User user = new User("John", "Smith", "jsmith@gmail.com");
session.setAttribute("user", user);
```

JSP code

```
<p>Hello ${user["firstName"]}</p>
```

An example that works with an array

Servlet code

```
String[] colors = {"Red", "Green", "Blue"};
ServletContext application = this.getServletContext();
application.setAttribute("colors", colors);
```

JSP code

```
<p>The first color is ${colors[0]}<br>
   The second color is ${colors[1]}
</p>
```

Another way to write the JSP code

```
<p>The first color is ${colors["0"]}<br>
   The second color is ${colors["1"]}
</p>
```

An example that works with a list

Servlet code

```
ArrayList<User> users = UserIO.getUsers(path);
session.setAttribute("users", users);
```

JSP code

```
<p>The first address on our list is ${users[0].email}<br>
   The second address on our list is ${users[1].email}
</p>
```

Another way to write the JSP code

```
<p>The first address on our list is ${users["0"].email}<br>
   The second address on our list is ${users["1"].email}
</p>
```

Description

- A *list* is a special type of collection such as an ArrayList<> that uses an index to retrieve an object that's stored in the collection.
- Although the [ ] operator can be used to work with JavaBeans and maps, it is commonly used to work with arrays and lists.
- With EL, the quotation marks are required for specifying a property in a JavaBean or a key in a map, but the quotation marks are optional when specifying an index of an array or a list.

Figure 8-4 How to use the [ ] operator to access arrays and lists

How to use the dot operator to access nested properties

The first example in figure 8-5 illustrates the most commonly used syntax for accessing nested properties. Here, the servlet code creates a Product object that has a property named code. Then, the servlet code stores this Product object in a LineItem object, and stores the LineItem object as a session attribute named item.

Since both the LineItem and Product classes follow the rules for JavaBeans, the JSP code can then use EL to retrieve the code property for the Product object that's stored in the item attribute. Although this figure only shows how to work with one nested property, there is no limit to the number of nested properties that you can access with the dot operator.

The second example shows that you can use the dot operator after the [ ] operator. Here, the only catch is that the object that's returned by the [ ] operator must be a JavaBean or a map. In this example, this works because the [ ] operator returns the same Product bean as the first example.

An example that accesses a nested property

Syntax

```
${attribute.property1.property2}
```

Servlet code

```
Product p = new Product();
p.setCode("pf01");
LineItem lineItem = new LineItem(p, 10);
session.setAttribute("item", lineItem);
```

JSP code

```
<p>Product code: ${item.product.code}</p>
```

Another way to access the same property

Syntax

```
${attribute["property1"].property2}
```

Servlet code

```
Product p = new Product();
p.setCode("pf01");
LineItem lineItem = new LineItem(p, 10);
session.setAttribute("item", lineItem);
```

JSP code

```
<p>Product code: ${item["product"].code}</p>
```

Description

- If a JavaBean has a property that returns another JavaBean, you can use the dot operator to access nested properties.
- There is no limit to the number of nested properties that you can access with the dot operator.

Figure 8-5 How to use the dot operator to access nested properties

Other skills for working with EL

Most of the time, the EL skills that you've learned so far are the ones that you use when you develop your JSPs. However, there are times when you may need some of the skills presented in the rest of this chapter.

How to use the [ ] operator to access attributes

The first example in figure 8-6 shows how you can use the [ ] operator to access an attribute. To start, the servlet code creates a map named usersMap that contains User objects, and it sets this map as an attribute. Although it isn't shown by this code, this map uses the email address as a key that can be used to get the associated User object. Then, the servlet code gets an email address from the request object, and it sets this string as an attribute named email.

Next, the JSP code retrieves the User object associated with the email attribute by coding the attribute within the [ ] operator. Since no quotation marks are coded around the attribute, EL attempts to evaluate the attribute. Here, it uses the value stored in the email attribute in the expression. In other words, if the email variable is "jsmith@gmail.com", the expression evaluates to:

```
${usersMap["jsmith@gmail.com"].firstName}
```

As a result, the User object that's mapped to "jsmith@gmail.com" is returned, and the first name for this user is displayed.

Note, however, that EL doesn't evaluate the variable within the [ ] operator if you code quotation marks around the emailAddress like this:

```
${usersMap["email"].firstName}
```

As a result, this expression returns an empty string.

The second example shows how you can use nested [ ] operators to access attributes. Here, the servlet code stores a map of User objects and an array of email addresses. Then, the JSP code uses nested [ ] operators to get the User object that's mapped to the first email address in the array of email addresses.

This works because the expression within the [ ] operator isn't enclosed in quotes. As a result, EL evaluates the expression as

```
${usersMap["jsmith@gmail.com"].firstName}
```

This means that the User object that's mapped to "jsmith@gmail.com" is returned, and the first name for this user is displayed. Here again, this example won't work if you enclose the expression within quotes.

An example that uses an attribute within the [ ] operator

Syntax

```
${attribute[attribute].property}
```

Servlet code

```
HashMap<String, User> usersMap = UserIO.getUsersMap(path);
session.setAttribute("usersMap", usersMap);

String email = request.getParameter("email");
session.setAttribute("email", email);
```

JSP code

```
<p>First name: ${usersMap[email].firstName}</p>
```

JSP code that returns an empty string

```
<!-- this doesn't work because the attribute is enclosed in quotes -->
<p>First name: ${usersMap["email"].firstName}</p>
```

Another example

Syntax

```
${attribute[attribute[index]].property}
```

Servlet code

```
HashMap<String, User> usersMap = UserIO.getUsersMap(path);
session.setAttribute("usersMap", usersMap);

String[] emails = {"jsmith@gmail.com", "joel@murach.com"};
session.setAttribute("emails", emails);
```

JSP code

```
<p>First name: ${usersMap[emails[0]].firstName}</p>
```

JSP code that returns an empty string

```
<!-- this doesn't work because the attribute is enclosed in quotes -->
<p>First name: ${usersMap["emails[0]"].firstName}</p>
```

Description

- If the expression within the [ ] operator isn't enclosed within quotes, EL evaluates the expression. To start, EL checks to see if the expression is an attribute. Then, it attempts to evaluate the expression.
- If multiple [ ] operators exist, the expression is evaluated from the innermost [ ] operator to the outermost [ ] operator. As a result, you can nest as many [ ] operators as necessary.

Figure 8-6 How to use the [ ] operator to access attributes

How to work with the other implicit EL objects

In figure 8-3, you learned how to use the implicit EL objects to specify scope. Now, figure 8-7 presents some other implicit EL objects that you can use to perform other tasks in a JSP. If you use the MVC pattern, though, you typically won't need to use these implicit objects.

The first example in this figure shows how to use the implicit objects for working with request parameters. Here, the HTML form has a parameter named firstName for the first name text box and a parameter named emailAddress for the next two text boxes. In other words, this form lets you enter one first name and two email addresses. Then, the JSP code shows how you can use the param object to get the value of the firstName parameter. In addition, this JSP code shows how you can use the paramValues parameter to get an array of strings that contains the values for the emailAddress parameter.

The second example shows how you can use the header object to get data from the HTTP header. Here, you can use the dot operator to get the value for any request headers that have a single word name. For instance, you can use the dot operator to get the value of the Accept header.

If a request header has more than one word in its name, you can use the [ ] operator to get its value. For instance, you can use this operator to get the value of the Accept-Encoding header. For now, don't worry if you don't understand how HTTP request headers work because you'll learn more about them in chapter 18.

When you learn more about HTTP request headers, you'll find that some HTTP request headers return a list of values. For example, the Accept header returns a list of MIME types that the browser can accept. If you want to return this list as an array, you can use the implicit headerValues object. Then, you can use the array that's returned to process these values.

The third example shows how to use the implicit cookie object to get the value of a cookie. Here, the servlet code creates a Cookie object with a name of emailCookie that stores an email address. Then, the JSP code uses the implicit cookie object to get the Cookie object, and it uses the value property of the cookie to get the email address that's stored within the cookie.

This works because the Cookie class follows all of the rules for a JavaBean, and it includes a method named getValue that can be used to get the value that's stored within the cookie. If necessary, you can use similar code to retrieve other properties of the cookie. For example, you can use the maxAge property to get the maximum age for the cookie.

Other implicit objects that you can use

EL implicit object	Description
`param`	A map that returns a value for the specified request parameter name.
`paramValues`	A map that returns an array of values for the specified request parameter name.
`header`	A map that returns the value for the specified HTTP request header. For a list of HTTP request headers, see chapter 18.
`headerValues`	A map that returns an array of values for the specified HTTP request header.
`cookie`	A map that returns the Cookie object for the specified cookie.
`initParam`	A map that returns the value for the specified parameter name in the context-param element of the web.xml file.
`pageContext`	A reference to the implicit pageContext object that's available from any JSP.

How to get parameter values from the request

An HTML form that has two parameters with the same name

```
<form action="emailList" method="post">
    <p>First name: <input type="text" name="firstName"></p>
    <p>Email address 1: <input type="text" name="email"></p>
    <p>Email address 2: <input type="text" name="email"></p>
</form>
```

JSP code

```
<p>First name: ${param.firstName}<br>
   Email address 1: ${paramValues.email[0]}<br>
   Email address 2: ${paramValues.email[1]}
</p>
```

How to get an HTTP header

JSP code

```
<p>MIME types: ${header.accept}<br>
   Compression types: ${header["accept-encoding"]}
</p>
```

The text that a browser might display

```
MIME types: text/html,application/xml;q=0.9,image/webp,*/*;q=0.8
Compression types: gzip,deflate,sdch
```

How to work with cookies

Servlet code

```
Cookie c = new Cookie("emailCookie", email);
c.setMaxAge(60*60);   //set its age to 1 hour
c.setPath("/");       //allow the entire application to access it
response.addCookie(c);
```

JSP code

```
<p>The email cookie: ${cookie.emailCookie.value}</p>
```

Figure 8-7 How to work with the implicit EL objects (part 1 of 2)

The fourth example in figure 8-7 shows how to use the initParam object to get a context initialization parameter. Here, the web.xml file uses the context-param element to store a context initialization parameter named custServEmail with a value of "custserv@murach.com". Then, the JSP code uses the initParam object to retrieve the value for this parameter. Note that this parameter is a context initialization parameter that's available to the entire web application, not a servlet initialization parameter that's only available to the current servlet.

The fifth example shows how to use the implicit pageContext object that's available from any JSP. Since the pageContext object follows the rules for a JavaBean, you can easily access any of its properties. In addition, the properties of the pageContext object allow you to access the objects for request, response, session, and application scope.

For example, you can use the request property to return an HttpRequest object that lets you get information about the current request. You can use the response property to return an HttpResponse object that lets you get information about the current response. You can use the session property to return an HttpSession object that lets you get information about the current session. And you can use the servletContext property to return a ServletContext object that lets you get information about the context for the application.

How to get a context initialization parameter

XML in the web.xml file

```
<context-param>
    <param-name>custServEmail</param-name>
    <param-value>custserv@murach.com</param-value>
</context-param>
```

JSP code

```
<p>The context init param: ${initParam.custServEmail}</p>
```

How to use the pageContext object

JSP code

```
<p>HTTP request method: ${pageContext.request.method}<br>
   HTTP response type: ${pageContext.response.contentType}<br>
   HTTP session ID: ${pageContext.session.id}<br>
   HTTP contextPath: ${pageContext.servletContext.contextPath}<br>
</p>
```

The text that a browser might display

```
HTTP request method: POST
HTTP response type: text/html
HTTP session ID: 4C1CFDB54B0339B53BE3AC8E9BADC0F5
HTTP servletContext path: /ch8email
```

Description

- The four implicit EL objects for specifying scope are presented in figure 8-3.
- All of the implicit objects used by EL are maps, except for the pageContext object, which is a JavaBean.

Figure 8-7 How to work with the implicit EL objects (part 2 of 2)

How to work with other EL operators

Earlier in this chapter, you learned how to use the dot operator and the [ ] operator that are available from EL. Now, figure 8-8 shows how to use some of the other operators that are available from EL. If necessary, you can use these operators to perform calculations and comparisons.

If you know how to code arithmetic, relational, and logical expressions in Java as described in *Murach's Java Programming*, you shouldn't have any trouble using these operators since they work similarly to the Java operators. Although you usually won't need to use these operators, they may come in handy from time to time.

You can use the arithmetic operators to perform mathematical calculations. The examples in this figure show that you can use scientific notation for extremely large or small numbers and that you can use parentheses to control or clarify the order of precedence.

These examples also show that EL treats any null values as zero. To illustrate, let's assume that you have an attribute named userID that can store an int value. Then, if the attribute stores an int value, EL uses the int value in the calculation. However, if the attribute stores a null value, EL uses zero in the calculation.

You can use the relational operators to compare two operands and return a true or false value. Although these operators work like the standard Java operators, you can use an alternate syntax that uses two letter combinations instead of symbols. For example, you can use eq instead of ==.

When you create relational expressions, you can use the null keyword to specify a null value, the true keyword to specify a true value, and the false keyword to specify a false value. In addition, when you create relational expressions, EL treats a null value as a false value. To illustrate, let's assume that you have an attribute named isDirty that stores a Boolean value. Then, if the attribute stores a true or false value, EL uses that value in the expression. However, if the attribute stores a null value, EL uses a false value for the isDirty attribute.

Arithmetic EL operators

Operator	Alternative	Description
+		Addition
-		Subtraction
*		Multiplication
/	div	Division
%	mod	Modulus (remainder)

Example	Result
${1+1}	2
${17.5+10}	27.5
${2.5E3}	2500.0
${2.5E3+10.4}	2510.4
${2-1}	1
${7*3}	21
${1 / 4}	0.25
${1 div 4}	0.25
${10 % 8}	2
${10 mod 8}	2
${1 + 2 * 4}	9
${(1 + 2) * 4}	12
${userID + 1}	9 if userID equals 8; 1 if userID equals 0

Relational operators

Operator	Alternative	Description
==	eq	Equal to
!=	ne	Not equal to
<	lt	Less than
>	gt	Greater than
<=	le	Less than or equal to
>=	ge	Greater than or equal to

Example	Result
${"s1" == "s1"}	true
${"s1" eq "s1"}	true
${1 == 1}	true
${1 != 1}	false
${1 ne 1}	false
${3 < 4}	true
${3 lt 4}	true
${3 > 4}	false
${3 gt 4}	false
${3 <= 4}	true
${3 >= 4}	false
${user.firstName == null}	true if firstName returns a null value
${user.firstName == ""}	true if firstName returns an empty string
${isDirty == true}	true if isDirty is true, false if isDirty is false, false if isDirty is null

Figure 8-8 How to work with the other EL operators (part 1 of 2)

You can use logical operators to combine multiple relational expressions and return a true or false value. The examples in this figure show how to use all three types of logical operators. You can use the And operator to specify that both relational expressions must be true for the entire expression to evaluate to true. You can use the Or operator to specify that at least one of the relational expressions must be true. And you can use the Not operator to reverse the value of the relational expression.

In addition, you may occasionally want to use the last two operators described in this figure. First, you may want to use the empty operator to check if a variable contains a null value or an empty string. If so, this operator returns a true value.

Second, you may want to use the ? and : operators to create a simple if statement. Here, you can code the condition for the if statement, followed by the ? operator, followed by the value that's returned for a true value, followed by the : operator, followed by the value that's returned for a false value. In the examples for these operators, the true and false keywords are used for the condition because this clearly shows how the ? and : operators work. However, you can substitute any relational or logical expression for the condition.

Logical operators

Operator	Alternative	Description
&&	and	And
||	or	Or
!	not	Not

Example	Result
`${"s1" == "s1" && 4 > 3}`	true
`${"s1" == "s1" and 4 > 3}`	true
`${"s1" == "s1" && 4 < 3}`	false
`${"s1" == "s1" || 4 < 3} true`	
`${"s1" != "s1" || 4 < 3} false`	
`${"s1" != "s1" or 4 < 3}`	false
`${!true}`	false
`${not true}`	false

Other operators

Syntax	Description
`empty x`	Returns true if the value of x is null or equal to an empty string.
`x ? y : z`	If x evaluates to true, returns y. Otherwise, returns z.

Example	Result
`${empty firstName}`	true if firstName returns a null value or an empty string
`${true ? "s1" : "s2"}`	s1
`${false ? "s1" : "s2"}`	s2

Keywords you can use in expressions

Keyword	Description
null	A null value
true	A true value
false	A false value

Description

- For arithmetic expressions, you can use parentheses to control or clarify the order of precedence.
- In arithmetic expressions, EL treats a null value as a zero.
- In logical expressions, EL treats a null value as a false value.

Figure 8-8 How to work with the other EL operators (part 2 of 2)

How to disable EL

For JSP 2.0 and later, the servlet container evaluates any code within the ${ and } characters as an EL expression. Most of the time, this is what you want. However, there's an outside chance that you may have one or more old JSPs that use the EL syntax for another purpose. In that case, you can disable EL as shown in figure 8-9.

To disable EL for a single JSP, you add a page directive to the JSP, and you set the isELIgnored attribute to true. Then, the servlet container won't evaluate any expressions that use the EL syntax. Instead, the servlet container passes this syntax on to the web browser, which typically causes the web browser to display the expression.

To disable EL for multiple pages, you can edit the jsp-config tag in the web.xml file for the application. This figure, for example, shows how to disable EL for all of the JSPs in the application. If necessary, though, you can modify the url-pattern element to disable EL only for selected JSPs within the application.

How to disable scripting

One of the benefits of EL is that it lets you remove JSP scripts from your JSPs, which makes it easier for web designers to work with them. In fact, once you learn how to use EL with JSTL as described in the next chapter, you can usually remove all scripting from your application.

When you're replacing old JSP scripts with EL and JSTL, it is sometimes hard to tell whether you've removed all scripting from your JSPs. Then, to check if you've done that, you can disable scripting for the entire application as described in this figure. After that, when you request a JSP that contains scripting, the servlet container displays an error page. In that case, you can edit the JSP to replace the scripting with EL, JSTL, or a combination of the two.

When you are developing new web applications, you may want to disable scripting from the start. This forces you (and any other programmers working on the application) to always use standard JSP tags, EL, and JSTL instead of scripting. Although this may require you to do more work up front to get the web application structured correctly with the MVC pattern, it should pay off in the long run by making your code easier to read and maintain.

How to disable EL

For a single page (a page directive)

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

For the entire application (the web.xml file)

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

How to disable scripting

For the entire application (the web.xml file)

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

Description

- For JSP 2.0 and later, the servlet container evaluates any text that starts with ${ and ends with } as an EL expression. Most of the time, this is what you want. If it isn't, you can ignore EL for a single JSP or for all JSPs in the entire application.
- If you want to remove all scripting from all JSPs in your application, you can modify the web.xml file so it doesn't allow scripting. Then, if you request a JSP that contains scripting, the servlet container displays an error page.

Figure 8-9 How to disable EL or scripting

Perspective

The goal of this chapter has been to show you how to use EL to get data from JavaBeans, maps, arrays, and lists that have been stored as an attribute in one of the four scopes of a web application. Along the way, you learned how to use some of the implicit EL objects and operators to perform some other useful tasks. Although there's more to learn about EL, this is probably all you need to know if you use the MVC pattern in your web applications.

When you use EL, you can remove much of the Java scripting and expressions from your JSPs, and that makes your code easier to read and maintain. That also makes it easier for web designers to work with these pages. To take this a step further, though, the next chapter shows you how to use the JSP Standard Tag Library (JSTL) to remove the rest of the scripting from your JSPs. As you'll see, JSTL tags work well with EL, and the two are often mixed in a JSP.

Summary

- The JSP *Expression Language* (*EL*) provides a compact syntax that lets you get data from JavaBeans, maps, arrays, and lists that have been stored as an attribute of a web application.
- A *JavaBean* is a special type of object that provides a standard way to access its *properties*.
- A *map* is a special type of collection that's used to store *key/value pairs*.
- A *list* is a type of collection that uses an *index* to retrieve an object that's stored in the collection.
- You can use the dot operator to work with JavaBeans and maps.
- You can use the [ ] operator to work with arrays and lists.
- If the expression within an [ ] operator isn't enclosed within quotes, EL evaluates the expression.
- If necessary, you can use the implicit EL objects to explicitly specify scope.
- You can use EL to get nested properties.
- You can use the implicit EL objects to work with request parameters, HTTP headers, cookies, context initialization parameters, and the implicit pageContext object that's available to all JSPs.
- You can use other EL operators to perform calculations and make comparisons.
- If necessary, you can disable EL or scripting for one or more pages in a web application.