

SIMPLIFYING ACCESS TO JAVA CODE: THE JSP 2.0 EXPRESSION LANGUAGE

Topics in This Chapter

- 
- Motivating use of the expression language
 - Invoking the expression language
 - Disabling the expression language
 - Preventing the use of classic scripting elements
 - Understanding the relationship of the expression language to the MVC architecture
 - Referencing scoped variables
 - Accessing bean properties, array elements, `List` elements, and `Map` entries
 - Using expression language operators
 - Evaluating expressions conditionally

Training courses from the book's author:

<http://courses.coreservlets.com/>

- *Personally* developed and taught by Marty Hall
- Available onsite at *your* organization (any country)
- Topics and pace can be customized for your developers
- Also available periodically at public venues
- Topics include Java programming, beginning/intermediate servlets and JSP, advanced servlets and JSP, Struts, JSF/MyFaces, Ajax, GWT, Ruby/Rails and more. Ask for custom courses!

Chapter 16

Training courses from the book's author:
<http://courses.coreservlets.com/>

- *Personally* developed and taught by Marty Hall
- Available onsite at *your* organization (any country)
- Topics and pace can be customized for your developers
- Also available periodically at public venues
- Topics include Java programming, beginning/intermediate servlets and JSP, advanced servlets and JSP, Struts, JSF/MyFaces, Ajax, GWT, Ruby/Rails and more. Ask for custom courses!

JSP 2.0 introduced a shorthand language for evaluating and outputting the values of Java objects that are stored in standard locations. This expression language (EL) is one of the two most important new features of JSP 2.0; the other is the ability to define custom tags with JSP syntax instead of Java syntax. This chapter discusses the expression language; Volume 2 discusses the new tag library capabilities.

Core Warning

The JSP expression language cannot be used in servers that support only JSP 1.2 or earlier.



For additional information, please see the section on the JSF EL in JSF tutorial at <http://coreservlets.com/>

16.1 Motivating EL Usage

As illustrated in Figure 16–1, there are a number of different strategies that JSP pages can use to invoke Java code. One of the best and most flexible options is the MVC approach (see Chapter 15). In that approach, a servlet responds to the request; invokes the appropriate business logic or data access code; places the resultant data in beans; stores the beans in the request, session, or servlet context; and forwards the request to a JSP page to present the result.

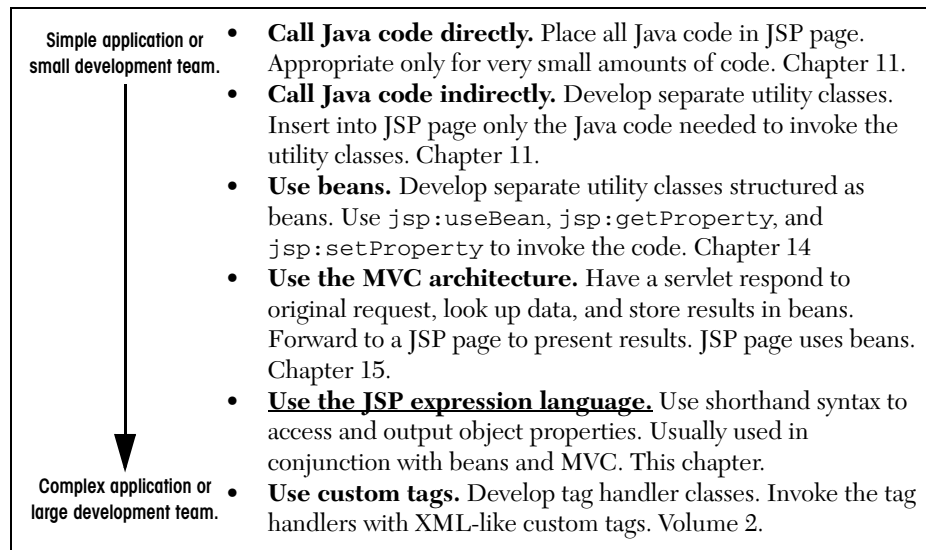


Figure 16-1 Strategies for invoking dynamic code from JSP.

The one inconvenient part about this approach is the final step: presenting the results in the JSP page. You normally use `jsp:useBean` and `jsp:getProperty`, but these elements are a bit verbose and clumsy. Furthermore, `jsp:getProperty` only supports access to simple bean properties; if the property is a collection or another bean, accessing the “subproperties” requires you to use complex Java syntax (precisely the thing you are often trying to avoid when using the MVC approach).

The JSP 2.0 expression language lets you simplify the presentation layer by replacing hard-to-maintain Java scripting elements or clumsy `jsp:useBean` and `jsp:getProperty` elements with short and readable entries of the following form.

```
${expression}
```

In particular, the expression language supports the following capabilities.

- **Concise access to stored objects.** To output a “scoped variable” (object stored with `setAttribute` in the `PageContext`, `HttpServletRequest`, `HttpSession`, or `ServletContext`) named `saleItem`, you use `${saleItem}`. See Section 16.5.
- **Shorthand notation for bean properties.** To output the `companyName` property (i.e., result of the `getCompanyName` method) of a scoped variable named `company`, you use `${company.companyName}`. To access the `firstName` property of the `president` property of a scoped variable named `company`, you use `${company.president.firstName}`. See Section 16.6.

- **Simple access to collection elements.** To access an element of an array, List, or Map, you use `${variable[indexOrKey]}`. Provided that the index or key is in a form that is legal for Java variable names, the dot notation for beans is interchangeable with the bracket notation for collections. See Section 16.7.
- **Succinct access to request parameters, cookies, and other request data.** To access the standard types of request data, you can use one of several predefined implicit objects. See Section 16.8.
- **A small but useful set of simple operators.** To manipulate objects within EL expressions, you can use any of several arithmetic, relational, logical, or empty-testing operators. See Section 16.9.
- **Conditional output.** To choose among output options, you do not have to resort to Java scripting elements. Instead, you can use `${test ? option1 : option2}`. See Section 16.10.
- **Automatic type conversion.** The expression language removes the need for most typecasts and for much of the code that parses strings as numbers.
- **Empty values instead of error messages.** In most cases, missing values or `NullPointerExceptions` result in empty strings, not thrown exceptions.

16.2 Invoking the Expression Language

In JSP 2.0, you invoke the expression language with elements of the following form.

```
${expression}
```

These EL elements can appear in ordinary text or in JSP tag attributes, provided that those attributes permit regular JSP expressions. For example:

```
<UL>
  <LI>Name: ${expression1}
  <LI>Address: ${expression2}
</UL>
<jsp:include page="${expression3}" />
```

When you use the expression language in tag attributes, you can use multiple expressions (possibly intermixed with static text) and the results are coerced to strings and concatenated. For example:

```
<jsp:include page="${expr1}blah${expr2}" />
```

This chapter will illustrate the use of expression language elements in ordinary text. Volume 2 of this book will illustrate the use of EL elements in attributes of tags that you write and tags that are provided by the JSP Standard Tag Library (JSTL) and JavaServer Faces (JSF) libraries.

Escaping Special Characters

If you want `${` to appear in the page output, use `\${` in the JSP page. If you want to use a single or double quote within an EL expression, use `\'` and `\"`, respectively.

16.3 Preventing Expression Language Evaluation

In JSP 1.2 and earlier, strings of the form `${ . . . }` had no special meaning. So, it is possible that the characters `${` appear within a previously created page that is now being used on a server that supports JSP 2.0. In such a case, you need to deactivate the expression language in that page. You have four options for doing so.

- **Deactivating the expression language in an entire Web application.** You use a `web.xml` file that refers to servlets 2.3 (JSP 1.2) or earlier. See the first following subsection for details.
- **Deactivating the expression language in multiple JSP pages.** You use the `jsp-property-group web.xml` element to designate the appropriate pages. See the second following subsection for details.
- **Deactivating the expression language in individual JSP pages.** You use the `isELEnabled` attribute of the page directive. See the third following subsection for details.
- **Deactivating individual expression language statements.** In JSP 1.2 pages that need to be ported unmodified across multiple JSP versions (with no `web.xml` changes), you can replace `$` with `$`, the HTML character entity for `$`. In JSP 2.0 pages that contain both expression language statements and literal `${` strings, you can use `\${` when you want `${` in the output.

Remember that these techniques are *only* necessary when the page contains the sequence `${`.

Deactivating the Expression Language in an Entire Web Application

The JSP 2.0 expression language is automatically deactivated in Web applications whose deployment descriptor (i.e., `WEB-INF/web.xml` file) refers to servlet specification version 2.3 or earlier (i.e., JSP 1.2 or earlier). The `web.xml` file is discussed in great detail in the second volume of the book, but this volume provides a quick introduction in Section 2.11 (Web Applications: A Preview). For example, the following empty-but-legal `web.xml` file is compatible with JSP 1.2, and thus indicates that the expression language should be deactivated by default.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
</web-app>
```

On the other hand, the following `web.xml` file is compatible with JSP 2.0, and thus stipulates that the expression language should be activated by default. (Both of these `web.xml` files, like *all* code examples presented in the book, can be downloaded from the book's source code archive at <http://www.coreservlets.com/>).

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
    "http://java.sun.com/xml/ns/j2ee web-app_2_4.xsd"
  version="2.4">
</web-app>
```

Deactivating the Expression Language in Multiple JSP Pages

In a Web application whose deployment descriptor specifies servlets 2.4 (JSP 2.0), you use the `el-ignored` subelement of the `jsp-property-group` `web.xml` element to designate the pages in which the expression language should be ignored. Here is an example that deactivates the expression language for all JSP pages in the `legacy` directory.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation=
           "http://java.sun.com/xml/ns/j2ee web-app_2_4.xsd"
         version="2.4">
  <jsp-property-group>
    <url-pattern>/legacy/*.jsp</url-pattern>
    <el-ignored>true</el-ignored>
  </jsp-property-group>
</web-app>
```

The `jsp-property-group` element is discussed in more detail in Volume 2 of this book.

Deactivating the Expression Language in Individual JSP Pages

To disable EL evaluation in an individual page, supply `false` as the value of the `isELEnabled` attribute of the page directive, as follows.

```
<%@ page isELEnabled="false" %>
```

Note that the `isELEnabled` attribute is new in JSP 2.0 and it is an error to use it in a server that supports only JSP 1.2 or earlier. So, you cannot use this technique to allow the same JSP page to run in either old or new servers without modification. Consequently, the `jsp-property-group` element is usually a better choice than the `isELEnabled` attribute.

Deactivating Individual Expression Language Statements

Suppose you have a JSP 1.2 page containing `{` that you want to use in multiple places. In particular, you want to use it in both JSP 1.2 Web applications and in Web applications that contain expression language pages. You want to be able to drop the page in any Web application without making *any* changes either to it or to the `web.xml` file. Although this is an unlikely scenario, it could happen, and none of the previously discussed constructs will serve the purpose. In such a case, you simply replace the `$` with the HTML character entity corresponding to the ISO 8859-1 value of `$` (36). So, you replace `{` with `${` throughout the page. For example,

```
&#36;{blah}
```

will portably display

```
${blah}
```

to the user. Note, however, that the character entity is translated to \$ by the *browser*, not by the *server*, so this technique will only work when you are outputting HTML to a Web browser.

Finally, suppose you have a JSP 2.0 page that contains both expression language statements and literal `${` strings. In such a case, simply put a backslash in front of the dollar sign. So, for example,

```
\${1+1} is ${1+1}.
```

will output

```
${1+1} is 2.
```

16.4 Preventing Use of Standard Scripting Elements

The JSP expression language provides succinct and easy-to-read access to scoped variables (Java objects stored in the standard locations). This capability eliminates much of the need for the explicit Java scripting elements described in Chapter 11. In fact, some developers prefer to use a no-classic-scripting-elements approach throughout their entire projects. You can use the `scripting-invalid` subelement of `jsp-property-group` to enforce this restriction. For example, the following `web.xml` file indicates that use of classic scripting elements in any JSP page will result in an error.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
    "http://java.sun.com/xml/ns/j2ee web-app_2_4.xsd"
  version="2.4">
  <jsp-property-group>
    <url-pattern>*.jsp</url-pattern>
    <scripting-invalid>true</scripting-invalid>
  </jsp-property-group>
</web-app>
```


16.5 Accessing Scoped Variables

When you use the MVC approach (Chapter 15), a servlet invokes code that creates the data, then uses `RequestDispatcher.forward` or `response.sendRedirect` to transfer control to the appropriate JSP page. To permit the JSP page to access the data, the servlet needs to use `setAttribute` to store the data in one of the standard locations: the `HttpServletRequest`, the `HttpSession`, or the `ServletContext`.

Objects in these locations are known as “scoped variables,” and the expression language has a quick and easy way to access them. You can also have scoped variables stored in the `PageContext` object, but this is much less useful because the servlet and the JSP page do not share `PageContext` objects. So, page-scoped variables apply only to objects stored earlier in the same JSP page, not to objects stored by a servlet.

To output a scoped variable, you simply use its name in an expression language element. For example,

```
${name}
```

means to search the `PageContext`, `HttpServletRequest`, `HttpSession`, and `ServletContext` (in that order) for an attribute named `name`. If the attribute is found, its `toString` method is called and that result is returned. If nothing is found, an empty string (not null or an error message) is returned. So, for example, the following two expressions are equivalent.

```
${name}  
<%= pageContext.findAttribute("name") %>
```

The problems with the latter approach are that it is verbose and it requires explicit Java syntax. It is possible to eliminate the Java syntax, but doing so requires the following even more verbose `jsp:useBean` code.

```
<jsp:useBean id="name" type="somePackage.SomeClass" scope="...">  
<%= name %>
```

Besides, with `jsp:useBean`, you have to know which scope the servlet used, and you have to know the fully qualified class name of the attribute. This is a significant inconvenience, especially when the JSP page author is someone other than the servlet author.

Choosing Attribute Names

To use the JSP expression language to access scoped variables, you must choose attribute names that would be legal as Java variable names. So, avoid dots, spaces, dashes, and other characters that are permitted in strings but forbidden in variable names.

Also, you should avoid using attribute names that conflict with any of the pre-defined names given in Section 16.8.

An Example

To illustrate access to scoped variables, the `ScopedVars` servlet (Listing 16.1) stores a `String` in the `HttpServletRequest`, another `String` in the `HttpSession`, and a `Date` in the `ServletContext`. The servlet forwards the request to a JSP page (Listing 16.2, Figure 16–2) that uses `${attributeName}` to access and output the objects.

Notice that the JSP page uses the same syntax to access the attributes, regardless of the scope in which they were stored. This is usually a convenient feature because MVC servlets almost always use unique attribute names for the objects they store. However, it is technically possible for attribute names to be repeated, so you should be aware that the expression language searches the `PageContext`, `HttpServletRequest`, `HttpSession`, and `ServletContext` *in that order*. To illustrate this, the servlet stores an object in each of the three shared scopes, using the attribute name repeated in all three cases. The value returned by `${repeated}` (see Figure 16–2) is the first attribute found when searching the scopes in the defined order (i.e., the `HttpServletRequest` in this case). Refer to Section 16.8 (Referencing Implicit Objects) if you want to restrict the search to a particular scope.

Listing 16.1 `ScopedVars.java`

```
package coreservlets;

/** Servlet that creates some scoped variables (objects stored
 *  as attributes in one of the standard locations). Forwards
 *  to a JSP page that uses the expression language to
 *  display the values.
 */

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

Listing 16.1 ScopedVars.java (*continued*)

```

public class ScopedVars extends HttpServlet {
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        request.setAttribute("attribute1", "First Value");
        HttpSession session = request.getSession();
        session.setAttribute("attribute2", "Second Value");
        ServletContext application = getServletContext();
        application.setAttribute("attribute3",
            new java.util.Date());
        request.setAttribute("repeated", "Request");
        session.setAttribute("repeated", "Session");
        application.setAttribute("repeated", "ServletContext");
        RequestDispatcher dispatcher =
            request.getRequestDispatcher("/el/scoped-vars.jsp");
        dispatcher.forward(request, response);
    }
}

```

Listing 16.2 scoped-vars.jsp

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>Accessing Scoped Variables</TITLE>
<LINK REL=STYLESHEET
    HREF="/el/JSP-Styles.css"
    TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
    <TR><TH CLASS="TITLE">
        Accessing Scoped Variables
    </TH>
</TR>
</TABLE>
<P>
<UL>
    <LI><B>attribute1:</B> ${attribute1}
    <LI><B>attribute2:</B> ${attribute2}
    <LI><B>attribute3:</B> ${attribute3}
    <LI><B>Source of "repeated" attribute:</B> ${repeated}
</LI>
</UL>
</BODY></HTML>

```

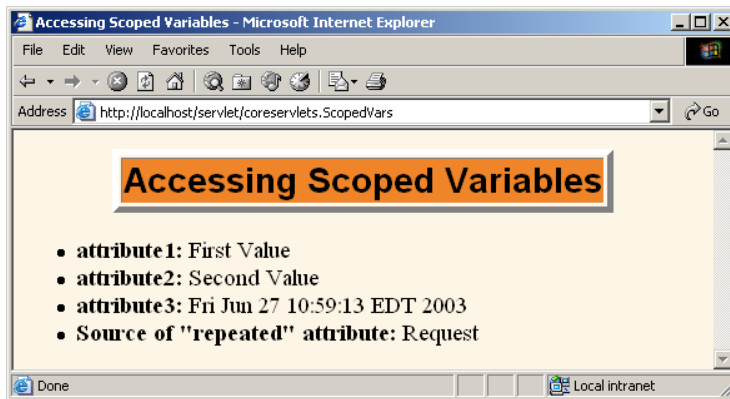


Figure 16–2 The JSP 2.0 expression language simplifies access to scoped variables: objects stored as attributes of the page, request, session, or servlet context.

16.6 Accessing Bean Properties

When you simply use `${name}`, the system finds the object named `name`, coerces it to a `String`, and returns it. Although this behavior is convenient, you rarely want to output *the actual object* that the MVC servlet stored. Rather, you typically want to output *individual properties* of that object.

The JSP expression language provides a simple but very powerful dot notation for accessing bean properties. To return the `firstName` property of a scoped variable named `customer`, you merely use `${customer.firstName}`. Although this form appears very simple, the system must perform reflection (analysis of the internals of the object) to support this behavior. So, assuming the object is of type `NameBean` and that `NameBean` is in the `coreservlets` package, to do the same thing with explicit Java syntax, you would have to replace

```
${customer.firstName}
```

with

```
<%@ page import="coreservlets.NameBean" %>
<%
NameBean person =
    (NameBean)pageContext.findAttribute("customer");
%>
<%= person.getFirstName() %>
```

Furthermore, the version with the JSP expression language returns an empty string if the attribute is not found, whereas the scripting element version would need additional code to avoid a `NullPointerException` in the same situation.

Now, JSP scripting elements are not the only alternative to use of the expression language. As long as you know the scope and fully qualified class name, you could replace

```
${customer.firstName}
```

with

```
<jsp:useBean id="customer" type="coreservlets.NameBean"
            scope="request, session, or application" />
<jsp:getProperty name="customer" property="firstName" />
```

However, the expression language lets you nest properties arbitrarily. For example, if the `NameBean` class had an `address` property (i.e., a `getAddress` method) that returned an `Address` object with a `zipCode` property (i.e., a `getZipCode` method), you could access this `zipCode` property with the following simple form.

```
${customer.address.zipCode}
```

There is no combination of `jsp:useBean` and `jsp:getProperty` that lets you do the same thing without explicit Java syntax.

Equivalence of Dot Notation and Array Notation

Finally, note that the expression language lets you replace dot notation with array notation (square brackets). So, for example, you can replace

```
${name.property}
```

with

```
${name["property"]}
```

The second form is rarely used with bean properties. However, it does have two advantages.

First, it lets you compute the name of the property at request time. With the array notation, the value inside the brackets can itself be a variable; with dot notation the property name must be a literal value.

Second, the array notation lets you use values that are illegal as property names. This is of no use when accessing bean properties, but it is very useful when accessing collections (Section 16.7) or request headers (Section 16.8).

An Example

To illustrate the use of bean properties, consider the `BeanProperties` servlet of Listing 16.3. The servlet creates an `EmployeeBean` (Listing 16.4), stores it in the request object with an attribute named `employee`, and forwards the request to a JSP page.

The `EmployeeBean` class has `name` and `company` properties that refer to `NameBean` (Listing 16.5) and `CompanyBean` (Listing 16.6) objects, respectively. The `NameBean` class has `firstName` and `lastName` properties and the `CompanyBean` class has `companyName` and `business` properties. The JSP page (Listing 16.7) uses the following simple EL expressions to access the four attributes:

```
${employee.name.firstName}
${employee.name.lastName}
${employee.company.companyName}
${employee.company.business}
```

Figure 16-3 shows the results.

Listing 16.3 `BeanProperties.java`

```
package coreservlets;

/** Servlet that creates some beans whose properties will
 *  be displayed with the JSP 2.0 expression language.
 */

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class BeanProperties extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        NameBean name = new NameBean("Marty", "Hall");
        CompanyBean company =
            new CompanyBean("coreservlets.com",
                           "J2EE Training and Consulting");
        EmployeeBean employee = new EmployeeBean(name, company);
        request.setAttribute("employee", employee);
        RequestDispatcher dispatcher =
            request.getRequestDispatcher("/el/bean-properties.jsp");
        dispatcher.forward(request, response);
    }
}
```

Listing 16.4 EmployeeBean.java

```
package coreservlets;

public class EmployeeBean {
    private NameBean name;
    private CompanyBean company;

    public EmployeeBean(NameBean name, CompanyBean company) {
        setName(name);
        setCompany(company);
    }

    public NameBean getName() { return(name); }

    public void setName(NameBean newName) {
        name = newName;
    }

    public CompanyBean getCompany() { return(company); }

    public void setCompany(CompanyBean newCompany) {
        company = newCompany;
    }
}
```

Listing 16.5 NameBean.java

```
package coreservlets;

public class NameBean {
    private String firstName = "Missing first name";
    private String lastName = "Missing last name";

    public NameBean() {}

    public NameBean(String firstName, String lastName) {
        setFirstName(firstName);
        setLastName(lastName);
    }

    public String getFirstName() {
        return(firstName);
    }
}
```

Listing 16.5 NameBean.java (*continued*)

```
public void setFirstName(String newFirstName) {
    firstName = newFirstName;
}

public String getLastName() {
    return(lastName);
}

public void setLastName(String newLastName) {
    lastName = newLastName;
}
}
```

Listing 16.6 CompanyBean.java

```
package coreservlets;

public class CompanyBean {
    private String companyName;
    private String business;

    public CompanyBean(String companyName, String business) {
        setCompanyName(companyName);
        setBusiness(business);
    }

    public String getCompanyName() { return(companyName); }

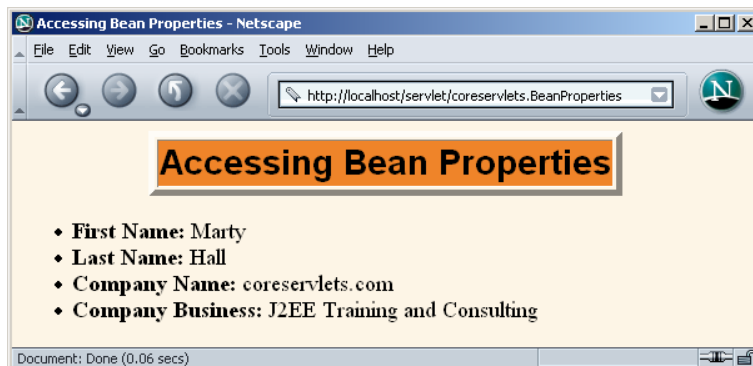
    public void setCompanyName(String newCompanyName) {
        companyName = newCompanyName;
    }

    public String getBusiness() { return(business); }

    public void setBusiness(String newBusiness) {
        business = newBusiness;
    }
}
```


Listing 16.7 bean-properties.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>Accessing Bean Properties</TITLE>
<LINK REL=STYLESHEET
      HREF="/el/JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
    Accessing Bean Properties
  </TH>
</TR>
</TABLE>
<P>
<UL>
  <LI><B>First Name:</B> ${employee.name.firstName}
  <LI><B>Last Name:</B>  ${employee.name.lastName}
  <LI><B>Company Name:</B> ${employee.company.companyName}
  <LI><B>Company Business:</B> ${employee.company.business}
</LI>
</UL>
</BODY></HTML>
```

**Figure 16–3** You use dots or array notation to access bean properties.

16.7 Accessing Collections

The JSP 2.0 expression language lets you access different types of collections in the same way: using array notation. For instance, if `attributeName` is a scoped variable referring to an array, `List`, or `Map`, you access an entry in the collection with the following:

```
${attributeName[entryName]}
```

If the scoped variable is an array, the entry name is the index and the value is obtained with `theArray[index]`. For example, if `customerNames` refers to an array of strings,

```
${customerNames[0]}
```

would output the first entry in the array.

If the scoped variable is an object that implements the `List` interface, the entry name is the index and the value is obtained with `theList.get(index)`. For example, if `supplierNames` refers to an `ArrayList`,

```
${supplierNames[0]}
```

would output the first entry in the `ArrayList`.

If the scoped variable is an object that implements the `Map` interface, the entry name is the key and the value is obtained with `theMap.get(key)`. For example, if `stateCapitals` refers to a `HashMap` whose keys are U.S. state names and whose values are city names,

```
${stateCapitals["maryland"]}
```

would return `"annapolis"`. If the `Map` key is of a form that would be legal as a Java variable name, you can replace the array notation with dot notation. So, the previous example could also be written as:

```
${stateCapitals.maryland}
```

However, note that the array notation lets you choose the key at request time, whereas the dot notation requires you to know the key in advance.

An Example

To illustrate the use of EL expressions to access collections, the `Collections` servlet (Listing 16.8) creates an array of strings, an `ArrayList`, and a `HashMap`. The servlet then forwards the request to a JSP page (Listing 16.9, Figure 16-4) that uses uniform array notation to access the elements of all three objects.

Purely numeric values are illegal as bean properties, so the array and `ArrayList` entries must be accessed with array notation. However, the `HashMap` entries could be accessed with either array or dot notation. We use array notation for consistency, but, for example,

```
${company["Ellison"]}
```

could be replaced with

```
${company.Ellison}
```

in Listing 16.9.

Listing 16.8 Collections.java

```
package coreservlets;

import java.util.*;

/** Servlet that creates some collections whose elements will
 *  be displayed with the JSP 2.0 expression language.
 *  <P>
 */

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Collections extends HttpServlet {
    public void doGet(HttpServletRequest request,
                     HttpServletResponse response)
        throws ServletException, IOException {
        String[] firstNames = { "Bill", "Scott", "Larry" };
        ArrayList lastNames = new ArrayList();
        lastNames.add("Ellison");
        lastNames.add("Gates");
        lastNames.add("McNealy");
        HashMap companyNames = new HashMap();
        companyNames.put("Ellison", "Sun");
        companyNames.put("Gates", "Oracle");
        companyNames.put("McNealy", "Microsoft");
        request.setAttribute("first", firstNames);
        request.setAttribute("last", lastNames);
        request.setAttribute("company", companyNames);
        RequestDispatcher dispatcher =
            request.getRequestDispatcher("/el/collections.jsp");
        dispatcher.forward(request, response);
    }
}
```

Listing 16.9 collections.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>Accessing Collections</TITLE>
<LINK REL=STYLESHEET
      HREF="/el/JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
    Accessing Collections
  </TH>
</TR>
</TABLE>
<P>
<UL>
  <LI>${first[0]} ${last[0]} (${company["Ellison"]})
  <LI>${first[1]} ${last[1]} (${company["Gates"]})
  <LI>${first[2]} ${last[2]} (${company["McNealy"]})
</LI>
</UL>
</BODY></HTML>
```

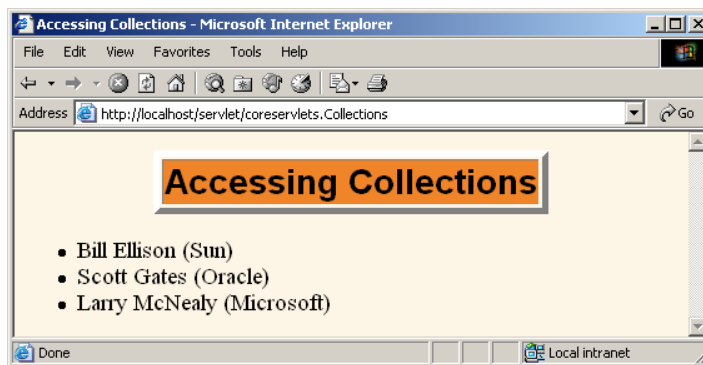


Figure 16–4 You use array notation or dots to access collections. Dots can be used only when the key is in a form that would be legal as a Java variable name.

16.8 Referencing Implicit Objects

In most cases, you use the JSP expression language in conjunction with the Model-View-Controller architecture (Chapter 15) in which the servlet creates the data and the JSP page presents the data. In such a scenario, the JSP page is usually only inter-

ested in the objects that the servlet created, and the general bean-access and collection-access mechanisms are sufficient.

However, the expression language is not restricted to use in the MVC approach; if the server supports JSP 2.0 and the `web.xml` file refers to servlets 2.4, the expression language can be used in any JSP page. To make this capability useful, the specification defines the following implicit objects.

pageContext

The `pageContext` object refers to the `PageContext` of the current page. The `PageContext` class, in turn, has `request`, `response`, `session`, `out`, and `servletContext` properties (i.e., `getRequest`, `getResponse`, `getSession`, `getOut`, and `getServletContext` methods). So, for example, the following outputs the current session ID.

```
${pageContext.session.id}
```

param and paramValues

These objects let you access the primary request parameter value (`param`) or the array of request parameter values (`paramValues`). So, for example, the following outputs the value of the `custID` request parameter (with an empty string, not `null`, returned if the parameter does not exist in the current request).

```
${param.custID}
```

For more information on dealing with request parameters, see Chapter 4.

header and headerValues

These objects access the primary and complete HTTP request header values, respectively. Remember that dot notation cannot be used when the value after the dot would be an illegal property name. So, for example, to access the `Accept` header, you could use either

```
${header.Accept}
```

or

```
${header["Accept"]}
```

But, to access the `Accept-Encoding` header, you must use

```
${header["Accept-Encoding"]}
```

For more information on dealing with HTTP request headers, see Chapter 5.

cookie

The `cookie` object lets you quickly reference incoming cookies. However, the return value is the `Cookie` object, not the cookie value. To access the value, use the standard `value` property (i.e., the `getValue` method) of the `Cookie` class. So, for example, either of the following outputs the value of the cookie named `userCookie` (or an empty string if no such cookie is found).

```
${cookie.userCookie.value}
${cookie["userCookie"].value}
```

For more information on using cookies, see Chapter 8.

initParam

The `initParam` object lets you easily access context initialization parameters. For example, the following outputs the value of the init param named `defaultColor`.

```
${initParam.defaultColor}
```

For more information on using initialization parameters, see Volume 2 of this book.

pageScope, requestScope, sessionScope, and applicationScope

These objects let you restrict where the system looks for scoped variables. For example, with

```
${name}
```

the system searches for `name` in the `PageContext`, the `HttpServletRequest`, the `HttpSession`, and the `ServletContext`, returning the first match it finds. On the other hand, with

```
${requestScope.name}
```

the system only looks in the `HttpServletRequest`.

An Example

The JSP page of Listing 16.10 uses the implicit objects to output a request parameter, an HTTP request header, a cookie value, and information about the server. Figure 16-5 shows the results.

Listing 16.10 implicit-objects.jsp

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>Using Implicit Objects</TITLE>
<LINK REL=STYLESHEET
      HREF="/el/JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
    Using Implicit Objects
  </TH>
</TR>
</TABLE>
<P>
<UL>
  <LI><B>test Request Parameter:</B> ${param.test}
  <LI><B>User-Agent Header:</B> ${header["User-Agent"]}
  <LI><B>JSESSIONID Cookie Value:</B> ${cookie.JSESSIONID.value}
  <LI><B>Server:</B> ${pageContext.servletContext.serverInfo}
</UL>
</BODY></HTML>

```

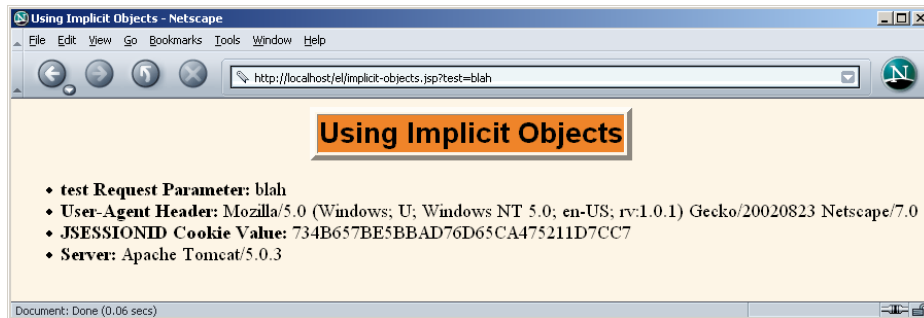


Figure 16–5 A number of scoped variables are defined automatically. These predefined variables are called “implicit objects.”

16.9 Using Expression Language Operators

The JSP 2.0 expression language defines a number of arithmetic, relational, logical, and missing-value-testing operators. Although use of the operators often results in code that is a bit shorter than the equivalent Java code, you should keep in mind the main purpose of the expression language: to provide concise access to existing objects, usually in the context of the MVC architecture. So, we see little benefit in replacing long and complex scripting elements with only slightly shorter expression language elements. Both approaches are wrong; complex business- or data-access-logic does not belong in JSP pages at all. As much as possible, restrict the use of the expression language to *presentation* logic; keep the business logic in normal Java classes that are invoked by servlets.

Core Approach

Use the expression language operators for simple tasks oriented toward presentation logic (deciding how to present the data). Avoid using the operators for business logic (creating and processing the data). Instead, put business logic in regular Java classes and invoke the code from the servlet that starts the MVC process.



Arithmetic Operators

These operators are typically used to perform simple manipulation of values already stored in beans. Resist using them for complex algorithms; put such code in regular Java code instead.

+ and –

These are the normal addition and subtraction operators, with two exceptions. First, if either of the operands is a string, the string is automatically parsed to a number (however, the system does *not* automatically catch `NumberFormatException`). Second, if either of the operands is of type `BigInteger` or `BigDecimal`, the system uses the corresponding `add` and `subtract` methods.

***, /, and div**

These are the normal multiplication and division operators, with a few exceptions. First, types are converted automatically as with the `+` and `-` operators. Second, the normal arithmetic operator precedence applies, so, for instance,

```
${ 1 + 2 * 3 }
```

returns 7, not 9. You can use parentheses to change the operator precedence. Third, the `/` and `div` operators are equivalent; both are provided for the sake of compatibility with both XPath and JavaScript (ECMAScript).

% and mod

The `%` (or equivalently, `mod`) operator computes the modulus (remainder), just as with `%` in the Java programming language.

Relational Operators

These operators are most frequently used with either the JSP expression language conditional operator (Section 16.10) or with custom tags whose attributes expect boolean values (e.g., looping tags like those in the JSP Standard Tag Library—JSTL—as discussed in Volume 2 of this book).

== and eq

These two equivalent operators check whether the arguments are equal. However, they operate more like the Java `equals` method than the Java `==` operator. If the two operands are the same object, `true` is returned. If the two operands are numbers, they are compared with Java `==`. If either operand is `null`, `false` is returned. If either operand is a `BigInteger` or `BigDecimal`, the operands are compared with `compareTo`. Otherwise, the operands are compared with `equals`.

!= and ne

These two equivalent operators check whether the arguments are different. Again, however, they operate more like the negation of the Java `equals` method than the Java `!=` operator. If the two operands are the same object, `false` is returned. If the two operands are numbers, they are compared with Java `!=`. If either operand is `null`, `true` is returned. If either operand is a `BigInteger` or `BigDecimal`, the operands are compared with `compareTo`. Otherwise, the operands are compared with `equals` and the opposite result is returned.

< and lt, > and gt, <= and le, >= and ge

These are the standard arithmetic operators with two exceptions. First, type conversions are performed as with `==` and `!=`. Second, if the arguments are strings, they are compared lexically.

Logical Operators

These operators are used to combine results from the relational operators.

&&, and, ||, or, !, not

These are the standard logical AND, OR, and NOT operators. They operate by coercing their arguments to `Boolean`, and they use the normal Java “short circuit” evaluation in which the testing is stopped as soon as the result can be determined. `&&` and `and` are equivalent, `||` and `or` are equivalent, and `!` and `not` are equivalent.

The empty Operator

empty

This operator returns `true` if its argument is `null`, an empty string, an empty array, an empty Map, or an empty collection. Otherwise it returns `false`.

An Example

Listing 16.11 illustrates several of the standard operators; Figure 16–6 shows the result.

Listing 16.11 operators.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>EL Operators</TITLE>
<LINK REL=STYLESHEET
      HREF="/el/JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
    EL Operators
  </TH>
</TR>
</TABLE>
```

Listing 16.11 operators.jsp (continued)

```

<P>
<TABLE BORDER=1 ALIGN="CENTER">
  <TR><TH CLASS="COLORED" COLSPAN=2>Arithmetic Operators
    <TH CLASS="COLORED" COLSPAN=2>Relational Operators
  <TR><TH>Expression<TH>Result<TH>Expression<TH>Result
  <TR ALIGN="CENTER">
    <TD>\${3+2-1}<TD>\${3+2-1} <!-- Addition/Subtraction -->
    <TD>\${1<2}<TD>\${1<2} <!-- Numerical comparison -->
  <TR ALIGN="CENTER">
    <TD>\${"1"+2}<TD>\${"1"+2} <!-- String conversion -->
    <TD>\${"a"<"b"}<TD>\${"a"<"b"} <!-- Lexical comparison -->
  <TR ALIGN="CENTER">
    <TD>\${1 + 2*3 + 3/4}<TD>\${1 + 2*3 + 3/4} <!-- Mult/Div -->
    <TD>\${2/3 > 3/2}<TD>\${2/3 > 3/2} <!-- >= -->
  <TR ALIGN="CENTER">
    <TD>\${3%2}<TD>\${3%2} <!-- Modulo -->
    <TD>\${3/4 == 0.75}<TD>\${3/4 == 0.75} <!-- Numeric = -->
  <TR ALIGN="CENTER">
    <!-- div and mod are alternatives to / and % -->
    <TD>\${(8 div 2) mod 3}<TD>\${(8 div 2) mod 3}
    <!-- Compares with "equals" but returns false for null -->
    <TD>\${null == "test"}<TD>\${null == "test"}

  <TR><TH CLASS="COLORED" COLSPAN=2>Logical Operators
    <TH CLASS="COLORED" COLSPAN=2><CODE>empty</CODE> Operator
  <TR><TH>Expression<TH>Result<TH>Expression<TH>Result
  <TR ALIGN="CENTER">
    <TD>\${(1<2) && (4<3)}<TD>\${(1<2) && (4<3)} <!--AND-->
    <TD>\${empty ""}<TD>\${empty ""} <!-- Empty string -->
  <TR ALIGN="CENTER">
    <TD>\${(1<2) || (4<3)}<TD>\${(1<2) || (4<3)} <!--OR-->
    <TD>\${empty null}<TD>\${empty null} <!-- null -->
  <TR ALIGN="CENTER">
    <TD>\${!(1<2)}<TD>\${!(1<2)} <!-- NOT -->
    <!-- Handles null or empty string in request param -->
    <TD>\${empty param.blah}<TD>\${empty param.blah}
</TABLE>
</BODY></HTML>

```

Arithmetic Operators		Relational Operators	
Expression	Result	Expression	Result
<code>\${3+2-1}</code>	4	<code>\${1<2}</code>	true
<code>\${"1"+2}</code>	3	<code>\${"a"<"b"}</code>	true
<code>\${1 + 2*3 + 3/4}</code>	7.75	<code>\${2/3 >= 3/2}</code>	false
<code>\${3%2}</code>	1	<code>\${3/4 == 0.75}</code>	true
<code>\${(8 div 2) mod 3}</code>	1.0	<code>\${null == "test"}</code>	false
Logical Operators		empty Operator	
Expression	Result	Expression	Result
<code>\${(1<2) && (4<3)}</code>	false	<code>\${empty ""}</code>	true
<code>\${(1<2) (4<3)}</code>	true	<code>\${empty null}</code>	true
<code>\${!(1<2)}</code>	false	<code>\${empty param.blah}</code>	true

Figure 16–6 The expression language defines a small set of operators. Use them with great restraint; invoke business logic from servlets, not from JSP pages.

16.10 Evaluating Expressions Conditionally

The JSP 2.0 expression language does not, in itself, provide a rich conditional evaluation facility. That capability is provided by the `c:if` and `c:choose` tags of the JSP Standard Tag Library (JSTL) or by another custom tag library. (JSTL and the creation of custom tag libraries are covered in Volume 2 of this book.)

However, the expression language supports the rudimentary `?:` operator as in the Java, C, and C++ languages. For example, if `test` evaluates to `true`,

```
${ test ? expression1 : expression2 }
```

returns the value of `expression1`; otherwise, it returns the value of `expression2`. Just remember that the main purpose of the expression language is to simplify presentation logic; avoid using this technique for business logic.

An Example

The servlet of Listing 16.12 creates two `SalesBean` objects (Listing 16.13) and forwards the request to a JSP page (Listing 16.14) for presentation (Figure 16–7). However, if the total sales number is negative, the JSP page wants to use a table cell with a red background. If it is positive, the page wants to use a white background. Implementing this behavior is a presentation task, so the `? :` operator is appropriate.

Listing 16.12 Conditionals.java

```
package coreservlets;

/** Servlet that creates scoped variables that will be used
 *  to illustrate the EL conditional operator (xxx ? xxx : xxx).
 */

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Conditionals extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        SalesBean apples =
            new SalesBean(150.25, -75.25, 22.25, -33.57);
        SalesBean oranges =
            new SalesBean(-220.25, -49.57, 138.25, 12.25);
        request.setAttribute("apples", apples);
        request.setAttribute("oranges", oranges);
        RequestDispatcher dispatcher =
            request.getRequestDispatcher("/el/conditionals.jsp");
        dispatcher.forward(request, response);
    }
}
```

Listing 16.13 SalesBean.java

```
package coreservlets;

public class SalesBean {
    private double q1, q2, q3, q4;

    public SalesBean(double q1Sales,
                     double q2Sales,
                     double q3Sales,
                     double q4Sales) {
        q1 = q1Sales;
        q2 = q2Sales;
        q3 = q3Sales;
        q4 = q4Sales;
    }

    public double getQ1() { return(q1); }

    public double getQ2() { return(q2); }

    public double getQ3() { return(q3); }

    public double getQ4() { return(q4); }

    public double getTotal() { return(q1 + q2 + q3 + q4); }
}
```

Listing 16.14 conditionals.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>Conditional Evaluation</TITLE>
<LINK REL=STYLESHEET
      HREF="/el/JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
    Conditional Evaluation
  </TH>
</TR>
</TABLE>
<P>
<TABLE BORDER=1 ALIGN="CENTER">
  <TR><TH>
    <TH CLASS="COLORED">Apples
    <TH CLASS="COLORED">Oranges
  </TH>
</TR>
</TABLE>
```

Listing 16.14 conditionals.jsp (*continued*)

```

<TR><TH CLASS="COLORED">First Quarter
    <TD ALIGN="RIGHT">${apples.q1}
    <TD ALIGN="RIGHT">${oranges.q1}
<TR><TH CLASS="COLORED">Second Quarter
    <TD ALIGN="RIGHT">${apples.q2}
    <TD ALIGN="RIGHT">${oranges.q2}
<TR><TH CLASS="COLORED">Third Quarter
    <TD ALIGN="RIGHT">${apples.q3}
    <TD ALIGN="RIGHT">${oranges.q3}
<TR><TH CLASS="COLORED">Fourth Quarter
    <TD ALIGN="RIGHT">${apples.q4}
    <TD ALIGN="RIGHT">${oranges.q4}
<TR><TH CLASS="COLORED">Total
    <TD ALIGN="RIGHT"
        bgcolor="${(apples.total < 0) ? "RED" : "WHITE" }">
        ${apples.total}
    <TD ALIGN="RIGHT"
        bgcolor="${(oranges.total < 0) ? "RED" : "WHITE" }">
        ${oranges.total}
</TABLE>
</BODY></HTML>

```

	Apples	Oranges
First Quarter	150.25	-220.25
Second Quarter	-75.25	-49.57
Third Quarter	22.25	138.25
Fourth Quarter	-33.57	12.25
Total	63.68	-119.32

Figure 16–7 You can use the C-style `? :` operator to conditionally output different elements. However, the JSP Standard Tag Library (JSTL) is often a better alternative for this kind of conditional operation.

16.11 Previewing Other Expression Language Capabilities

This chapter summarizes most of the capabilities of the JSP 2.0 expression language. However, there are two capabilities that we did not discuss here: the use of the expression language in tag libraries and the use of expression language functions. We postpone coverage of these two capabilities to Volume 2 because they require techniques covered only in the second volume of the book.

Quick preview: expression language elements can be used in any custom tag attribute that allows request time expressions (i.e., whose `attribute` element specifies `true` for `rtexprvalue`). Functions correspond to public `static` methods in regular Java classes and are defined with the `function` element of the Tag Library Descriptor (TLD) file.