The JSTL Expression Language

Topics in This Chapter

- Expression Language Overview
- Expressions
- Identifiers
- Operators
- Type Coercion
- Literal Values
- Implicit Objects
- Method Invocation
- EL Expressions in Custom Actions
- Common Mistakes

Chapter 2

Although JSTL, as its name implies, provides a set of standard tags, its single most important feature may well be the expression language it defines. That expression language greatly reduces the need for specifying tag attribute values with Java code and significantly simplifies accessing all sorts of application data, including beans, arrays, lists, maps, cookies, request parameters and headers, context initialization parameters, and so on. In fact, the JSTL expression language adds so much value to JSP that it will be incorporated into JSP 2.0.[1]

This chapter examines the JSTL expression language in detail, starting with expressions and identifiers and ending with sections on using the expression language for custom action attributes and common mistakes that developers make when using the expression language.

Note: To illustrate the JSTL expression language, this chapter uses a number of JSTL actions, such as <c:out>, <c:if>, and <c:forEach>, that have not yet been formally discussed in this book. However, the use of those actions is intuitive and this chapter does not use any of those action's advanced features. See Chapter 3, "General-Purpose and Conditional Actions," and "Iteration Actions" on page 150 for formal discussions of the actions used throughout this chapter.

1. The JSP expert group will do everything possible to ensure that the JSP 2.0 expression language is backward-compatible with the JSTL 1.0 expression language.

2.1 Expression Language Overview

The JSTL expression language is a simple language inspired by ECMAScript (also known as JavaScript) and XPath. The expression language provides:

- Expressions and identifiers
- Arithmetic, logical, and relational operators
- Automatic type coercion
- Access to beans, arrays, lists, and maps
- Access to a set of implicit objects and servlet properties

All of the features listed above are described in this chapter.

Throughout this book, for convenience the expression language is referred to with the acronym EL and JSTL expressions are referred to as EL expressions.

How the Expression Language Works

Nearly all of the JSTL actions have one or more dynamic attributes that you can specify with an EL expression;[2] for example, you can specify a request parameter with the <c:out> action's `value` attribute like this:

```
<c:out value='${param.emailAddress}'/>
```

The preceding expression displays the value of a request parameter named `emailAddress`. You can also use EL expressions to perform conditional tests, for example:

```
<c:if test='${not empty param.emailAddress}'>...</c:if>
```

The body of the preceding <c:if> action is evaluated if the `emailAddress` request parameter is not empty, meaning neither `null` nor an empty string.

If you're using JSTL with JSP 1.2, you can only use JSTL expressions to specify values for JSTL action attributes, as illustrated above.[3] All JSTL actions that have dynamic attributes interpret EL expressions before they are passed to the action's tag handler, so the expression language is applied—and values are typically coerced—before the tag handler gets them.

2. Dynamic values for JSTL actions from the runtime (RT) library can be specified as JSP expressions.
3. Starting with JSP 2.0, you will be able to use EL expressions in JSP template text. See "Expressions" on page 41 for more information.

How to Use the Expression Language

Attributes of JSTL actions can be specified with EL expressions in one of three ways. First, an attribute can be specified with a single expression like this:[4]

```
<jstl:action value='${expr}'/>
```

In the preceding code fragment, the expression `${expr}` is evaluated and its value is coerced to the type expected by the `value` attribute.

Attribute values can also be specified as strings, like this:

```
<jstl:action value='text'/>
```

The string specified for the `value` attribute in the preceding code fragment is coerced to the type expected by that attribute.

Finally, attribute values can consist of one or more expressions intermixed with strings, like this:

```
<jstl:action value='${expr}text${expr}${expr}more text${expr}'/>
```

In the previous code fragment, each of the four expressions is evaluated in order from left to right, coerced to a string, and concatenated with the intermixed text. The resulting string is subsequently coerced to the value expected by the `value` attribute.

2.2 Expressions

EL expressions are invoked with this syntax: `${expr}`, where `expr` represents an expression. Expressions can consist of:

- Identifiers—see "Identifiers" on page 43
- Binary and unary operators—see "Operators" on page 44
- String, boolean, integer, and floating-point literals and `null`—see "Literal Values" on page 63
- Implicit objects, such as `param`, `cookie`, or `header`—see "Implicit Objects" on page 64

4. The action <jstl:action> represents any JSTL action.

Until JSP 2.0, when the JSTL expression language is scheduled to be incorporated into the JSP specification, you can only use EL expressions to specify attributes of JSTL actions; for example, the following code fragment from the Database Actions chapter specifies an SQL data source as a comma-separated string constructed with four EL expressions and three commas:[5]

```
<sql:setDataSource dataSource='${url},${driver},${user},${pwd}'
                        scope='session'/>
```

If you upgrade to JSP 2.0, you can have EL expressions in template text; for example, you will be able to execute an SQL query like this:[6]

```
<%-- This only works with containers that support JSP 2.0 --%>

<sql:query>
   ${customerQuery}
</sql:query>
```

The `customerQuery` scoped variable referenced by the EL expression in the preceding code fragment is a string that specifies a particular query.

Until JSP 2.0, you are restricted to using EL expressions to specify attributes for JSTL actions; for example, you can still execute the query listed in the preceding code fragment like this:

```
<%-- This works with containers that support JSP 1.2 --%>

<sql:query sql='${customerQuery}'/>
```

Alternatively, you could use the <c:out> action if you had to specify the query in the body of the action like this:

```
<%-- This also works with containers that support JSP 1.2 --%>

<sql:query>
   <c:out value='${customerQuery}'/>
</sql:query>
```

The <c:out> action in the preceding code fragment sends the string stored in the `customerQuery` scoped variable to the current `JspWriter`, which points to the <sql:query> action's body, so the preceding code fragment is functionally equivalent to the two preceding code fragments.

5. See "Specify Your Data Source with <sql:setDataSource>" on page 369.
6. The JSP 2.0 specification is scheduled to be completed in late 2002.

2.3 Identifiers

Identifiers in the expression language represent the names of objects stored in one of the JSP scopes: page, request, session, or application. Those types of objects are referred to throughout this book as *scoped variables*.

When the expression language encounters an identifier, it searches for a scoped variable with that name in the page, request, session, and application scopes, in that order; for example, the following code fragment stores a string in page scope and accesses that string with an EL expression:

```
<% // Create a string
   String s = "Richard Wilson";

   // Store the string in page scope
   pageContext.setAttribute("name", s); %>

<%-- Access the string with an EL expression --%>
<c:out value='${name}'/>
```

In the preceding code fragment, the expression `${name}` resolves to a reference to the string named `name` that was placed in page scope by the scriptlet. That reference is specified for the <c:out> action's `value` attribute. When the <c:out> action is confronted with an object reference for its `value` attribute, it coerces that object to a string by invoking its `toString` method, which in this case produces the value of the string, so the output of the preceding code fragment is `Richard Wilson`. If the `name` string had been placed in a different scope, the expression `${name}` would still resolve to that string, as long as there was not another object with the same name in another scope that was searched first. For example, if two objects named `name` are stored in request and application scopes, the expression `${name}` would resolve to the object stored in request scope.[7]

Identifiers must adhere to the syntax for Java programming language identifiers; for example, you cannot use characters such as - or / in an identifier.

The two sections that follow—"Accessing JavaBeans Components" and "Accessing Objects Stored in Arrays, Lists, and Maps" on page 52—illustrate how to use identifiers to access beans and collections, respectively, that are stored in JSP scopes.

7. Storing beans that have the same name in different scopes is not recommended because the JSP specification allows one of those beans to override the other.

2.4 Operators

JSTL offers a small set of operators, listed in Table 2.1.

Table 2.1 Expression Language Operators

Type	Operators
Arithmetic	`+ - * / (div) % (mod)`
Grouping	`()`
Identifier Access	`. []`
Logical	`&& (and) || (or) ! (not) empty`
Relational	`== (eq) != (ne) < (lt) > (gt) <= (le) >= (ge)`
Unary	-

You need to know three things about the operators in Table 2.1. First, you need to know the operators' syntax; for example, the + operator is used like this: `A + B`. That material is not covered here because you use EL expressions just like you use their Java equivalents. The second thing you need to know is operator precedence, so you can deduce that `1 + 3 * 5` is `16` but `(1 + 3) * 5` is `20`. Operator precedence is discussed in "Operator Precedence" on page 45. The third thing you need to know about the operators listed in Table 2.1 is what data types they prefer and how the EL performs type coercion. The former is briefly discussed here and the latter is discussed in "Type Coercion" on page 62.

All of the binary arithmetic operations prefer `Double` values and all of them will resolve to `0` if either of their operands is `null`. The operators `+ - * %` will try to coerce their operands to `Long` values if they cannot be coerced to `Double`.

The grouping operators, which are parentheses, can be used to force operator precedence, as discussed above. The identifier access operators are discussed in "The . and [] Operators" on page 45, so that discussion is not repeated here.

The relational operators all have textual equivalents; for example, either == or `eq` will suffice for the equality operator. Those equivalents are provided for XML generation. Like binary arithmetic operations, all of the relational operators prefer `Double` values but will make do with `Long` values if the operands cannot be converted to `Double`.

The logical operators prefer to work with `Boolean` operands. You can use the `empty` operator to see if a value is either `null` or an empty string (`""`). That operator comes in handy when you are interpreting request parameters.

Operator Precedence

The precedence for EL operators is listed below:

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

The operators are listed above from left to right and top to bottom according to precedence; for example, the `[]` operator has precedence over the `.` operator, and the modulus (`%` or `mod`) operator, which represents a division remainder, has precedence over the logical operators.

The `.` and `[]` Operators

The JSTL expression language provides two operators—`.` and `[]`—that let you access scoped variables and their properties. The `.` operator is similar to the Java `.` operator, but instead of invoking methods, you access bean properties; for example, if you have a `Name` bean stored in a scoped variable named `name` and that bean contains `firstName` and `lastName` properties, you can access those properties like this:

```
First Name: <c:out value='${name.firstName}'/>
Last Name: <c:out value='${name.lastName}'/>
```

Assuming that there is a bean named `name` that has readable properties `firstName` and `lastName` in one of the four JSP scopes—meaning methods named `getFirstName` and `getLastName`—the preceding code fragment will display those properties.

You can also use the `[]` operator to access bean properties; for example, the preceding code fragment could be rewritten like this:

```
First Name: <c:out value='${name["firstName"]}'/>
Last Name: <c:out value='${name["lastName"]}'/>
```

The `[]` operator is a generalization of the `.` operator, which is why the two previous code fragments are equivalent, but the `[]` operator lets you specify a *computed value*. "A Closer Look at the [] Operator" on page 56 takes a closer look at how the `[]` operator works.

You can also use the `[]` operator to access objects stored in maps, lists, and arrays; for example, the following code fragment accesses the first object in an array:

```
<% String[] array = { "1", "2", "3" };
   pageContext.setAttribute("array", array); %>

<c:out value='${array[0]}'/>
```

The preceding code fragment creates an array of strings and stores it in page scope with a scriptlet. Subsequently, the <c:out> action accesses the first item in the array with `${array[0]}`.

The following sections—"Accessing JavaBeans Components" and "Accessing Objects Stored in Arrays, Lists, and Maps" on page 52—explore in greater detail the use of the `.` and `[]` operators to access bean properties and objects stored in collections.

Accessing JavaBeans Components

This section shows you how to use the `.` and `[]` operators to access bean properties, including nested beans. Listing 2.1, Listing 2.2, and Listing 2.3 list the implementation of three beans: `Name`, `Address`, and `UserProfile`.

The preceding beans are simple JavaBean components. The `Name` bean has two properties: `firstName` and `lastName`. The `Address` bean has four properties: `streetAddress`, `city`, `state`, and `zip`. The `UserProfile` bean has two properties: `name` and `address`. `UserProfile` beans contain references to `Name` and `Address` beans.

Figure 2–1 shows a JSP page that creates a user profile and accesses its properties with EL expressions.

Listing 2.1 *WEB-INF/classes/beans/Name.java*

```
package beans;

public class Name {
   private String firstName, lastName;

   // JavaBean accessors for first name
   public void setFirstName(String firstName) {
      this.firstName = firstName;
   }
   public String getFirstName() {
      return firstName;
   }

   // JavaBean accessors for last name
   public void setLastName(String lastName) {
      this.lastName = lastName;
   }
   public String getLastName() {
      return lastName;
   }
}
```

Listing 2.2 *WEB-INF/classes/beans/Address.java*

```
package beans;

public class Address {
   private String streetAddress, city, state;
   private int zip;

   // JavaBean accessors for street address
   public void setStreetAddress(String streetAddress) {
      this.streetAddress = streetAddress;
   }
   public String getStreetAddress() {
      return streetAddress;
   }

   // JavaBean accessors for city
   public void setCity(String city) {
      this.city = city;
   }
```

Listing 2.2 *WEB-INF/classes/beans/Address.java (cont.)*

```
 public String getCity() {
      return city;
   }
// JavaBean accessors for state
   public void setState(String state) {
      this.state = state;
   }
   public String getState() {
      return state;
   }

   // JavaBean accessors for zip
   public void setZip(int zip) {
      this.zip = zip;
   }
   public int getZip() {
      return zip;
   }
}
```

Listing 2.3 *WEB-INF/classes/beans/UserProfile.java*

```
package beans;

public class UserProfile {
   private Name name;
   private Address address;

   // JavaBean accessors for name
   public void setName(Name name) {
      this.name = name;
   }
   public Name getName() {
      return name;
   }

   // JavaBean accessors for address
   public void setAddress(Address address) {
      this.address = address;
   }
   public Address getAddress() {
      return address;
   }
}
```

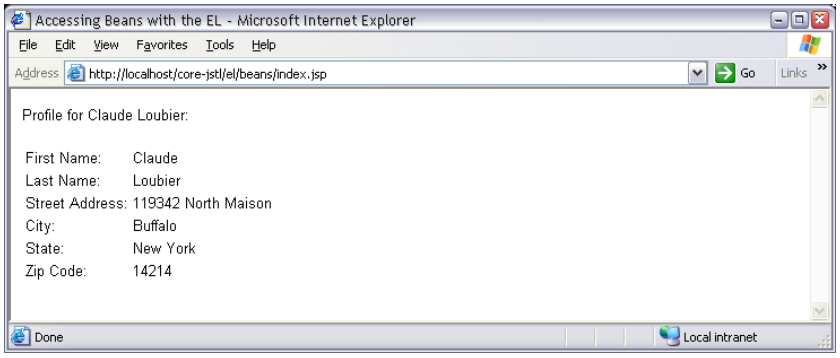

Figure 2–1 Accessing Beans with the Expression Language

The JSP page shown in Figure 2–1 is listed in Listing 2.4.

Listing 2.4 *Accessing JavaBean Properties*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
   <head>
      <title>Accessing Beans with the EL</title>
   </head>
   <body>
      <%@ taglib uri='http://java.sun.com/jstl/core' prefix='c' %>

      <%-- Create a Name bean and store it in page scope --%>
      <jsp:useBean id='name' class='beans.Name'>
         <%-- Set properties with strings --%>
         <jsp:setProperty name='name'
                      property='firstName' value='Claude'/>
         <jsp:setProperty name='name'
                      property='lastName'  value='Loubier'/>
      </jsp:useBean>

<%-- Create an Address bean and store it in page scope --%>
      <jsp:useBean id='address' class='beans.Address'>
         <%-- Set properties with strings --%>
         <jsp:setProperty name='address'
                      property='streetAddress'
                         value='119342 North Maison'/>
         <jsp:setProperty name='address'
                      property='city'  value='Buffalo'/>
<jsp:setProperty name='address'
                      property='state' value='New York'/>
```

Listing 2.4 *Accessing JavaBean Properties (cont.)*

```
<jsp:setProperty name='address'
                      property='zip'   value='14214'/>
      </jsp:useBean>

      <%-- Create a UserProfile bean and store it in
           page scope --%>
      <jsp:useBean id='profile'
                class='beans.UserProfile'>
         <%-- Set properties with the name bean and address
              bean stored in page scope --%>
         <jsp:setProperty name='profile'
            property='name'
               value='<%= (beans.Name)
                           pageContext.getAttribute("name") %>'/>

         <jsp:setProperty name='profile'
            property='address'
               value='<%= (beans.Address)
                          pageContext.getAttribute("address") %>'/>
      </jsp:useBean>

      <%-- Show profile information --%>

      Profile for
      <%-- Access the name bean's firstName property directly,
           without specifying scope --%>
      <c:out value='${name["firstName"]}'/>

      <%-- Access the name bean's lastName property through the
           profile bean by explicitly specifying scope --%>
      <c:out value='${pageScope.profile.name.lastName}'/>:
      <p>
      <table>
         <tr>
            <%-- Access the UserProfile bean's properties without
                 explicitly specifying scope --%>
            <td>First Name:</td>
            <td><c:out value='${profile["name"].firstName}'/></td>
         </tr><tr>
            <td>Last Name:
            <td><c:out value='${profile.name["lastName"]}'/></td>
         </tr><tr>
            <td>Street Address:
            <td><c:out value='${profile.address.streetAddress}'/>
            </td>
```

Listing 2.4 *Accessing JavaBean Properties (cont.)*

```
</tr><tr>
            <td>City:
            <td><c:out value='${profile.address.city}'/></td>
         </tr><tr>
            <td>State:
            <td><c:out value='${profile.address.state}'/></td>
         </tr><tr>
            <td>Zip Code:
            <td><c:out value='${profile.address.zip}'/></td>
         </tr>
      </table>
   </body>
</html>
```

The preceding JSP page creates three beans: a name bean, an address bean, and a user profile bean; the name and address beans are used to create the user profile. All three beans are stored in page scope.

The JSP page listed in Listing 2.4 uses EL expressions to access properties of the user profile. First, the JSP page accesses the name bean's `firstName` property with the expression `${name["firstName"]}`, which is equivalent to this expression: `${name.firstName}`.

Next, the JSP page accesses the name bean's `lastName` property with this expression: `${pageScope.profile.name.lastName}`. The expression starts with the `pageScope` identifier, which is an implicit object that provides access to all page-scoped attributes.[8] The user profile bean—named `profile`—that exists in page scope is accessed by name with an identifier, and its enclosed `name` bean is also accessed with an identifier. Finally, the `lastName` property of that `name` bean is accessed with another identifier.

The rest of the JSP page accesses the `profile` bean's properties by using the `.` and `[]` operators. Remember that the `.` and `[]` operators are interchangeable when accessing bean properties, so the expression `${profile["name"].firstName}` is equivalent to `${profile.name.firstName}` and `${profile.name["lastName"]}` is equivalent to `${profile.name.lastName}`.

Now that we've seen how to access bean properties, let's see how to access objects stored in arrays, lists, and maps.

8. See "Implicit Objects" on page 64 for more information about the JSTL implicit objects.

Accessing Objects Stored in Arrays, Lists, and Maps

In "Accessing JavaBeans Components" on page 46 we discussed a Web application that created user profiles and accessed their properties, all in a single JSP page. For a change of pace, this section discusses a Web application that creates user profiles in a servlet and accesses their properties in a JSP page.[9] Figure 2–2 shows the Web application's JSP page.

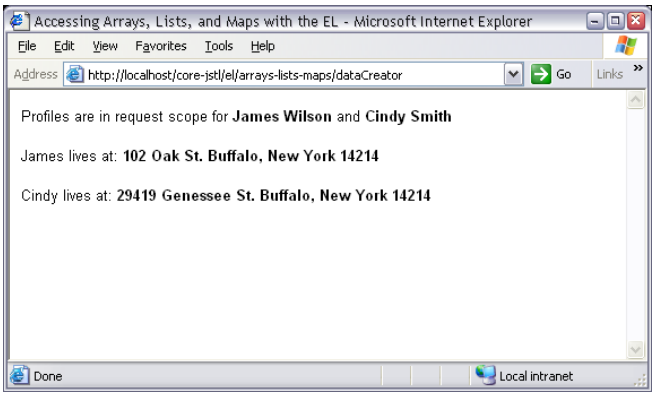

Figure 2–2 Accessing Arrays, Lists, and Maps with the JSTL Expression Language

The JSP page shown in Figure 2–2 is accessed indirectly with the URL `/dataCreator`. That URL invokes a servlet that creates user profiles and forwards to the JSP page. Listing 2.5 lists the application's deployment descriptor, which maps the `/dataCreator` URL to the `dataCreator` servlet.

The `dataCreator` servlet is listed in Listing 2.6.

The preceding servlet creates two user profiles and stores those profiles in an array, a map, and a list. Subsequently, the servlet stores the array, map, and list in request scope and forwards to a JSP page named `showData.jsp`. That JSP page is shown in Figure 2–2 and listed in Listing 2.7.

As the preceding JSP page illustrates, you access objects in an array with the `[]` operator, just as you would in Java by specifying a `0`-based index into the array; for example, the expression `${profileArray[0].name.firstName}` accesses the first name of the first profile stored in the `profileArray`.

9. Creating application data in a servlet separates the model from its presentation, which results in more flexible and extensible software.

Listing 2.5 *WEB-INF/web.xml*

```
<?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/j2ee/dtds/web-app_2.3.dtd">

<web-app>
   <servlet>
      <servlet-name>dataCreator</servlet-name>
      <servlet-class>DataCreatorServlet</servlet-class>
   </servlet>

   <servlet-mapping>
      <servlet-name>dataCreator</servlet-name>
      <url-pattern>/dataCreator</url-pattern>
   </servlet-mapping>
</web-app>
```

Listing 2.6 *WEB-INF/classes/DataCreatorServlet.java*

```
import java.io.IOException;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import beans.*;

public class DataCreatorServlet extends HttpServlet {
   public void doGet(HttpServletRequest request,
                     HttpServletResponse response)
                     throws IOException, ServletException {
      // Create an array, map, and list
      UserProfile[] profileArray = createProfiles();
      HashMap         profileMap = new HashMap();
      LinkedList     profileList = new LinkedList();

      // Populate the list and map
      for(int i=0; i < profileArray.length; ++i) {
         UserProfile profile = profileArray[i];

profileList.add(profile);

         String firstName = profile.getName().getFirstName(),
                lastName = profile.getName().getLastName(),
                     key = firstName + " " + lastName;
```

Listing 2.6 *WEB-INF/classes/DataCreatorServlet.java (cont.)*

```
profileMap.put(key, profile);
      }

      // Store the array, map, and list in request scope
      request.setAttribute("profileArray", profileArray);
      request.setAttribute("profileMap",   profileMap);
      request.setAttribute("profileList",  profileList);

      // Forward the request and response to /showData.jsp
      RequestDispatcher rd =
        getServletContext().getRequestDispatcher("/showData.jsp");

      rd.forward(request, response);
   }
   private UserProfile[] createProfiles() {
      // Create an array of user profiles
      UserProfile[] userProfiles = {
         new UserProfile(new Name("James", "Wilson"),
                         new Address("102 Oak St.", "Buffalo",
                                     "New York", 14214)),
         new UserProfile(new Name("Cindy", "Smith"),
                         new Address("29419 Genessee St.",
                                     "Buffalo", "New York", 14214))
      };
      return userProfiles;
   }
}
```

You access objects stored in a list with the same syntax used for accessing objects in arrays; for example, in the preceding JSP page, the expression `${profileList[1].name.firstName}` accesses the first name of the second profile stored in the `profileList`.

The Java class libraries offer quite a few different types of maps, including hash tables, hash maps, attributes, and tree maps. All of those data structures store *key/value* pairs of objects. To access those objects using the EL, you specify a key, enclosed in double quotes, with the `[]` operator; for example, the JSP page listed in Listing 2.7 accesses Cindy Smith's last name with this expression: `${profileMap["Cindy Smith"].name.lastName}`.

As you can tell from the JSP pages listed in Listing 2.4 on page 49 and Listing 2.7, accessing nested bean properties can be rather verbose, although it's much more succinct than accessing properties with a JSP expression. You can reduce that verbosity by creating page-scoped variables that directly reference beans stored in

Listing 2.7 *showData.jsp*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
   <head>
      <title>Accessing Arrays, Lists, and Maps with the EL</title>
   </head>
   <body>
      <%@ taglib uri='http://java.sun.com/jstl/core' prefix='c' %>

      <%-- Access the first and last names stored in the two
           user profiles through the array, list, and map --%>
      Profiles are in request scope for <b>
       <c:out value='${profileArray[0].name.firstName}'/>
       <c:out value='${profileArray[0].name.lastName}'/></b>
      and <b>
       <c:out value='${profileList[1].name.firstName}'/>
       <c:out value='${profileMap["Cindy Smith"].name.lastName}'/>
       </b><p>

      <%-- Store the two profiles in page-scoped variables --%>
      <c:set var='jamesProfile'
           value='${profileMap["James Wilson"]}'/>

      <c:set var='cindyProfile'
           value='${profileList[1]}'/>

      <%-- Show address information, through the page-scoped
           variables --%>
      <c:out value='${jamesProfile.name.firstName}'/> lives at:<b>
      <c:out value='${jamesProfile.address.streetAddress}'/>
      <c:out value='${jamesProfile.address.city}'/>,
      <c:out value='${jamesProfile.address.state}'/>
      <c:out value='${jamesProfile.address.zip}'/></b>

      <p>

      <c:out value='${cindyProfile.name.firstName}'/> lives at:<b>
      <c:out value='${cindyProfile.address.streetAddress}'/>
      <c:out value='${cindyProfile.address.city}'/>,
      <c:out value='${cindyProfile.address.state}'/>
      <c:out value='${cindyProfile.address.zip}'/>

   </body>
</html>
```

data structures. For example, the JSP page listed in Listing 2.7 stores James Wilson's user profile in a page-scoped variable named `jamesProfile`. That JSP page also creates a page-scoped variable named `cindyProfile` that directly references the user profile for Cindy Smith. Those page-scoped variables make it easier to access the user profiles; for example, instead of accessing Cindy's first name through the profile map like this: `${profileMap["Cindy Smith"].name.firstName}`, you can access it like this: `${cindyProfile.name.firstName}`.

In this and the preceding section, we discussed how to use the `[]` operator to access bean properties and beans stored in arrays, lists, and maps. In the next section, we take a closer look at how the `[]` operator works and why you might prefer that operator to the `.` operator when accessing beans.

A Closer Look at the `[]` Operator

As discussed in "Accessing JavaBeans Components" on page 46 and "Accessing Objects Stored in Arrays, Lists, and Maps" on page 52, you use the `[]` operator with this syntax: `${`*`identifier`*`[`*`subexpression`*`]}`. Here's how expressions with that syntax are evaluated:

1. Evaluate the `identifier` and the `subexpression`; if either resolves to `null`, the expression is `null`.
2. If the identifier is *a bean*: The `subexpression` is coerced to a `String` value and that string is regarded as a name of one of the bean's properties. The expression resolves to the value of that property; for example, the expression **`${name.["lastName"]}`** translates into the value returned by **`name.getLastName()`**.
3. If the identifier is *an array*: The `subexpression` is coerced to an `int` value—which we'll call `subexpression-int`—and the expression resolves to `identifier[subexpression-int]`. For example, for an array named `colors`, **`colors[3]`** represents the fourth object in the array. Because the subexpression is coerced to an `int`, you can also access that color like this: **`colors["3"]`**; in that case, JSTL coerces `"3"` into `3`. *That feature may seem like a very small contribution to JSTL, but because request parameters are passed as strings, it can be quite handy.*
4. If the identifier is *a list*: The `subexpression` is also coerced to an `int`—which we will also call `subexpression-int`—and the expression resolves to the value returned from `identifier.get(subexpression-int)`, for example: **`colorList[3]`** and **`colorList["3"]`** both resolve to the fourth element in the list.

5. If the identifier is *a map*: The `subexpression` is regarded as one of the map's keys. That expression is not coerced to a value because map keys can be any type of object. The expression evaluates to `identifier.get(subexpression)`, for example, **`colorMap[Red]`** and **`colorMap["Red"]`**. The former expression is valid only if a scoped variable named `Red` exists in one of the four JSP scopes and was specified as a key for the map named `colorMap`.

Table 2.2 lists the methods that the EL invokes on your behalf.

Table 2.2 Methods That the EL Invokes for You

Identifier Type	Example Use	Method Invoked
JavaBean component	`${colorBean.red}` `${colorBean["red"]}`	`colorBean.getRed()`
`Array`	`${colorArray[2]}` `${colorArray["2"]}`	`Array.get(colorArray, 2)`
`List`	`colorList[2]` `colorList["2"]`	`colorList.get(2)`

`Map`	`colorMap[red]`
	
	`colorMap["red"]`
		`colorMap.get(pageContext.findAttribute("red"))`
		
		`colorMap.get("red")`

JSTL developers rely heavily on maps because the EL provides 11 indispensable implicit objects, of which 10 are maps. Everything, from request parameters to cookies, is accessed through a map. Because of this reliance on maps, you need to understand the meaning of the last row in Table 2.2. You access a map's values through its keys, which you can specify with the `[]` operator, for example, in Table 2.2, `${colorMap[red]}` and `${colorMap["red"]}`. The former specifies an *identifier* for the key, whereas the latter specifies a *string*. For the identifier, the `PageContext.findAttribute` method searches all four JSP scopes for a scoped variable with the name that you specify, in this case, `red`. On the other hand, if you specify a string, it's passed directly to the map's `get` method.

The [] Operator's Special Ability

Although it may not be obvious from our discussion so far, the `[]` operator has a special ability that its counterpart, the `.` operator, does not have—it can operate on an *expression*, whereas the `.` operator can only operate on an *identifier*. For example, you can do this: `${colorMap[param.selectedColor]}`, which uses the

string value of the `selectedColor` request parameter as a key for a map named `colorMap`.[10] That's something that you can't do with the `.` operator.

Figure 2–3 shows a Web application that uses the `[]` operator's special ability to show request header values.

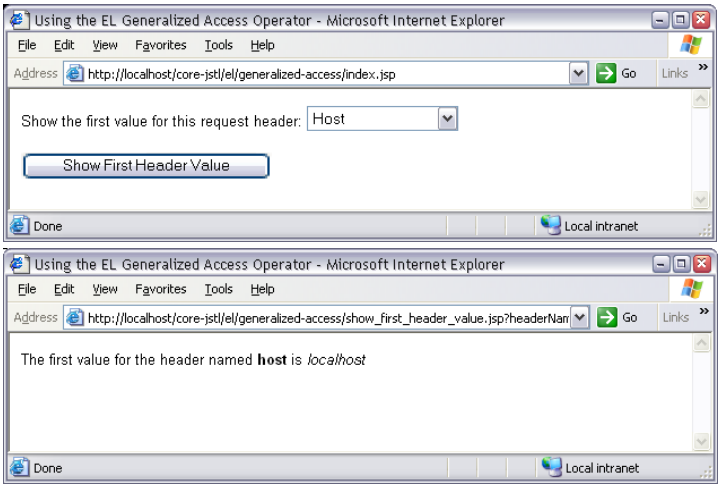

Figure 2–3 Illustrating an Advantage of the `[]` Operator

The top picture in Figure 2–3 shows a JSP page that lets you select a request header. After you activate the `Show First Header Value` button, the JSP page shown in the bottom picture shows the first value for the request header that was selected.

The JSP page shown in the top picture in Figure 2–3 is listed in Listing 2.8.

The preceding JSP page uses the `header` JSTL implicit object to iterate over request headers. The names of those request headers are used to create an HTML `select` element named `headerName`. The `select` element resides in a form whose action is `show_first_header_value.jsp`, so that the JSP page is loaded when you activate the `Show First Header Value` button. That JSP page is listed in Listing 2.9.

The preceding JSP page uses two JSTL implicit objects: `param`, which is a map of request parameters, and `header`, which is a map of request headers. The subexpression `param.headerName` accesses the `headerName` request parameter's value, and the expression `${header[param.headerName]}` resolves to the first value for that request header.

10. The `param` implicit object lets you access request parameters; see "Implicit Objects" on page 64 for more information.

Listing 2.8 *Selecting a Request Header*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
   <head>
      <title>Using the EL Generalized Access Operator</title>
   </head>
   <body>
      <%@ taglib uri='http://java.sun.com/jstl/core' prefix='c' %>

      <form action='show_first_header_value.jsp'>
         Show the first value for this request header:

         <select name='headerName'>
            <c:forEach var='hdr' items='${header}'>
               <option value='<c:out value="${hdr.key}"/>'>
                  <c:out value='${hdr.key}'/>
               </option>
            </c:forEach>
         </select>

         <p><input type='submit' value='Show First Header Value'/>
      </form>
   </body>
</html>
```

Listing 2.9 *Using the `[]` Operator with a Request Parameter*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
   <head>
      <title>Using the EL Generalized Access Operator</title>
   </head>
   <body>
      <%@ taglib uri='http://java.sun.com/jstl/core' prefix='c' %>

      The first value for the header named <b>
      <c:out value='${param.headerName}'/></b> is <i>
      <c:out value='${header[param.headerName]}'/></i>
   </body>
</html>
```

The `empty` Operator

Testing for the existence of request parameters can be tricky because they evaluate to `null` if they don't exist but they evaluate to an empty string (`""`) if their value was not specified. Most of the time, when you check for the existence of request parameters, you don't have to distinguish the former from the latter; you just want to know whether a value was specified. For that special task, you can use the `empty` operator, which tests whether an identifier is `null` or doesn't exist, as illustrated by the Web application shown in Figure 2–4.

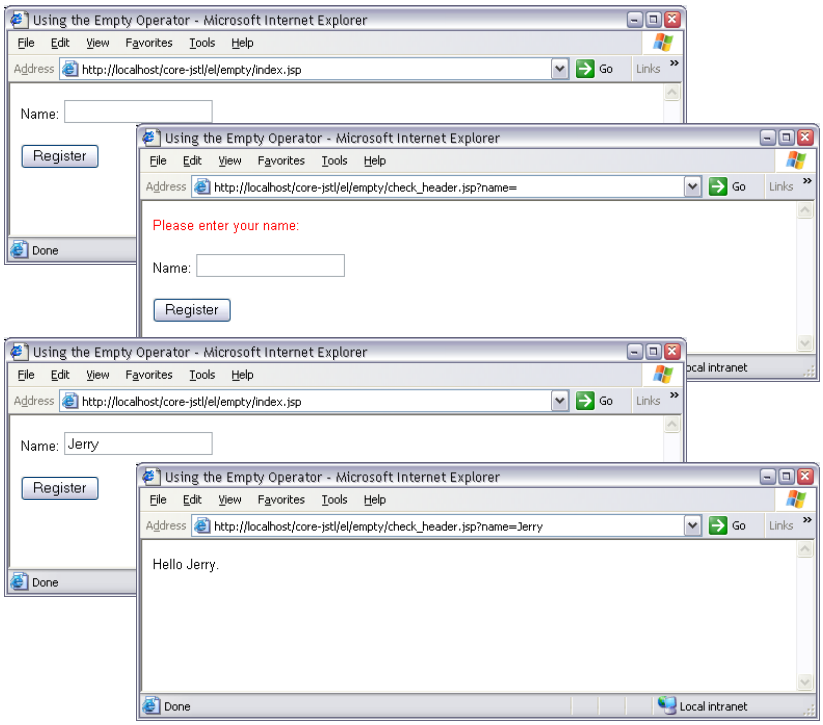

Figure 2–4 Using the `empty` Operator to Test for the Existence of Request Parameters

The Web application shown in Figure 2–4 consists of two JSP pages, one that lets you enter a name and another that checks for a corresponding request parameter. As illustrated by the top two pictures in Figure 2–4, if you don't enter anything in the `name` field, the latter JSP page prints an error message and includes the referring JSP page. The bottom two pictures illustrate successful access to the `name` request parameter. The JSP page with the `name` input field is listed in Listing 2.10.

The preceding JSP page includes form.jsp, which is listed in Listing 2.11.

The action for the form in the preceding JSP page is `check_header.jsp`, which is listed in Listing 2.12.

Listing 2.10 *index.jsp*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
   <head>
      <title>Using the Empty Operator</title>
   </head>
   <body>
      <jsp:include page='form.jsp'/>
   </body>
</html>
```

Listing 2.11 *form.jsp*

```
<form action='check_header.jsp'>
   Name: <input type='text' name='name'/>
   <p><input type='submit' value='Register'/>
</form>
```

Listing 2.12 *check_header.jsp*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
   <head>
      <title>Using the Empty Operator</title>
   </head>
   <body>
      <%@ taglib uri='http://java.sun.com/jstl/core' prefix='c' %>

      <c:choose>
         <c:when test='${not empty param.name}'>
            Hello <c:out value='${param.name}'/>.
         </c:when>

         <c:otherwise>
            <font color='red'>
               Please enter your name:<p>
            </font>

            <jsp:include page='form.jsp'/>
         </c:otherwise>
      </c:choose>
   </body>
</html>
```

The preceding JSP page combines the `not` and `empty` operators to see whether the `name` request parameter was specified; if so it displays a personalized greeting; otherwise, it prints an error message and includes the referring JSP page.

2.5 Type Coercion

The EL defines a comprehensive set of coercion rules for various data types. Those rules are summarized in Table 2.3.

Table 2.3 JSTL Type Coercion[a]

convert to —>	`Boolean`	`Character`	`Number`	`String`
`Boolean`	——	ERROR	ERROR	*x*`.toString()`
`Character`	ERROR	——	`(short)x`	*x*`.toString()`
`Number`	ERROR	`(char)x`	——	*x*`.toString()`
`String` (not empty)	`Boolean.valueOf(x)`	*x*`.charAt(0)`	*N*`.valueOf(x)`	——
Other	ERROR	ERROR	ERROR	*x*`.toString()`
`null`	`Boolean.false`	`(char)0`	`0`	`""`
`""`	`Boolean.false`	`(char)0`	`0`	`""`

a. *x* represents the object being converted, *N* represents a `Number` subclass, and `""` represents an empty string

In the preceding table, types in the left column are converted into the types specified in the table's header. For example, if you specify an action attribute's value as a string and that attribute's type is `Character`, the EL will convert that string to the first character in the string by invoking the method `x.charAt(0)`, where `x` represents the string. Likewise, strings are coerced to `Boolean` values with the `static Boolean.valueOf(x)`, where `x` represents the string.

Table 2.3 also shows how `null` values and empty strings are converted into booleans, characters, numbers, and strings. JSTL actions typically avoid throwing exceptions because the coercions shown in Table 2.3 are applied to their attribute values before they receive them.

If you specify a `null` value or an empty string in an expression, the EL's coercion rules ensure that sensible default values are used instead; for example:

```
<c:out value='${count + param.increment}'/>
```

In the preceding code fragment, the expression specified for the `value` attribute coerces a *string* (a request parameter value named `increment`) to an *integer* which is added to the `count` scoped variable and sent to the current `JspWriter` by the <c:out> action.

If the `increment` request parameter does not exist, `param.increment` resolves to `null`. If it exists, but no value was specified for it—perhaps it represents an HTML `input` element in a form that was left blank—it resolves to an empty string. Either way, as you can see from Table 2.3, the EL coerces the string value of `param.increment` to `0`, and the expression `${count + param.increment}` evaluates to the value of the `count` scoped variable.

In general, JSTL actions avoid throwing exceptions, instead favoring sensible default values like `0` for `null` and empty strings.

Another thing you don't have to worry about is throwing a `null` pointer exception if you try to access an identifier that is `null`; for example, the expression `${userProfile.address.city}` resolves to `null` if `userProfile`, `address` or `city` is `null` because the EL coerces that value into one of the appropriate values in Table 2.3.

2.6 Literal Values

The JSTL expression language defines boolean, integer, floating-point, string, and `null` literals, as shown in Table 2.4.

Table 2.4 JSTL Literal Values

Type	Examples
`Boolean`	`Boolean.true Boolean.false`
`Integer`	`143 +3 -4 2435`
`Double`	`1.43 -2.35 2.34E9`
`String`	`"string in double quotes"` `'a string in single quotes'`
`Null`	`null`

2.7 Implicit Objects

Arguably, the most useful feature of the JSTL expression language is the implicit objects it defines for accessing all kinds of application data. Those implicit objects are listed in Table 2.5.

Table 2.5 JSTL Implicit Objects

Implicit Object	Type	Key[a]	Value
`cookie`	`Map`	Cookie name	Cookie
`header`	`Map`	Request header name	Request header value
`headerValues`	`Map`	Request header name	`String[]` of request header values
`initParam`	`Map`	Initialization parameter name	Initialization parameter value
`param`	`Map`	Request parameter name	Request parameter value
`paramValues`	`Map`	Request parameter name	`String[]` of request parameter values
`pageContext`	`PageContext`	N/A	N/A
`pageScope`	`Map`	Page-scoped attribute name	Page-scoped attribute value
`requestScope`	`Map`	Request-scoped attribute name	Request-scoped attribute value
`sessionScope`	`Map`	Session-scoped attribute name	Session-scoped attribute value
`applicationScope`	`Map`	Application-scoped attribute name	Application-scoped attribute value

a. All keys are strings.

There are three types of JSTL implicit objects:

- Maps for a single set of values, such as request headers and cookies:

  `param`, `paramValues`, `header`, `headerValues`, `initParam`, `cookie`

- Maps for scoped variables in a particular scope:

  `pageScope`, `requestScope`, `sessionScope`, `applicationScope`

- The page context: `pageContext`

The rest of this section examines each of the JSTL implicit objects in the order listed above; the first category begins at "Accessing Request Parameters" below, the second category begins at "Accessing Scoped Attributes" on page 78, and use of the `pageContext` implicit object begins at "Accessing JSP Page and Servlet Properties" on page 80.

Accessing Request Parameters

Request parameters are the lifeblood of most Web applications, passing information from one Web component to another. That crucial role makes the `param` and `paramValues` implicit objects, both of which access request parameters, the most heavily used JSTL implicit objects.

The `param` and `paramValues` implicit objects are both maps of request parameters. For both the `param` and `paramValues` maps, keys are request parameter names, but the values corresponding to those keys are different for `param` and `paramValues`; `param` stores the *first value* specified for a request parameter, whereas `paramValues` stores a `String` array that contains *all the values* specified for a request parameter.[11]

Most often, the overriding factor that determines whether you use `param` or `paramValue` is the type of HTML element a request parameter represents; for example, Figure 2–5 shows a Web application that uses both `param` and `paramValues` to display request parameters defined by a form.

11. Two of the implicit objects listed in Table 2.5 have plural names: `paramValues` and `headerValues`; both are maps that associate keys with `String` arrays. The other implicit objects associate keys with scalar values.

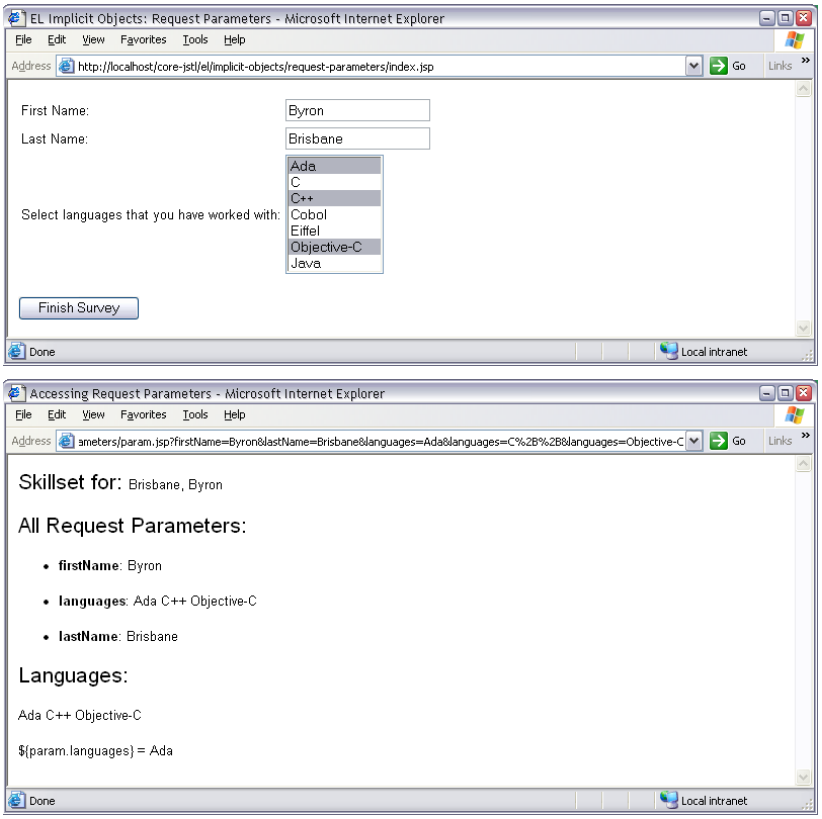

Figure 2–5 Accessing Request Parameters with the `param` and `paramValues` Implicit Objects

The Web application shown in Figure 2–5 consists of two JSP pages, one that contains a form (top picture) and another that interprets the form's data (bottom picture). Listing 2.13 lists the JSP page that contains the form.

Listing 2.13 *index.jsp*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
   <head>
      <title>EL Implicit Objects: Request Parameters</title>
   </head>

   <body>
      <form action='param.jsp'>
```

Listing 2.13 *index.jsp (cont.)*

```
<table>
            <tr>
               <td>First Name:</td>
               <td><input type='text' name='firstName'/></td>
            </tr>
            <tr>
               <td>Last Name:</td>
               <td><input type='text' name='lastName'/></td>
            </tr>
            <tr>
               <td>
                  Select languages that you have worked with:
               </td>
               <td>
                  <select name='languages' size='7'
                      multiple='true'>
                     <option value='Ada'>Ada</option>
                     <option value='C'>C</option>
                     <option value='C++'>C++</option>
                     <option value='Cobol'>Cobol</option>
                     <option value='Eiffel'>Eiffel</option>
                     <option value='Objective-C'>
                        Objective-C
                     </option>
                     <option value='Java'>Java</option>
                  </select>
               </td>
            </tr>
         </table>
         <p><input type='submit' value='Finish Survey'/>
      </form>
   </body>
</html>
```

The preceding JSP page is unremarkable; it creates an HTML form with two textfields and a `select` element that allows multiple selection. That form's action, `param.jsp`, is the focus of our discussion. It is listed in Listing 2.14.

Listing 2.14 *param.jsp*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
   <head>
      <title>Accessing Request Parameters</title>
```

Listing 2.14 *param.jsp (cont.)*

```
 </head>
<body>
      <%@ taglib uri='http://java.sun.com/jstl/core' prefix='c' %>

      <font size='5'>
         Skillset for:
      </font>

      <%-- Access the lastName and firstName request parameters
           parameters by name --%>
      <c:out value='${param.lastName}'/>,
      <c:out value='${param.firstName}'/>

      <%-- Show all request parameters and their values --%>
      <p><font size='5'>
         All Request Parameters:
      </font><p>

      <%-- For every String[] item of paramValues... --%>
      <c:forEach var='parameter' items='${paramValues}'>
         <ul>
            <%-- Show the key, which is the request parameter
                 name --%>
            <li><b><c:out value='${parameter.key}'/></b>:</li>

            <%-- Iterate over the values -- a String[] --
                 associated with this request parameter --%>
            <c:forEach var='value' items='${parameter.value}'>
               <%-- Show the String value --%>
               <c:out value='${value}'/>
            </c:forEach>
         </ul>
      </c:forEach>

      <%-- Show values for the languages request parameter --%>
      <font size='5'>
         Languages:
      </font><p>

      <%-- paramValues.languages is a String [] of values for the
           languages request parameter --%>
      <c:forEach var='language' items='${paramValues.languages}'>
         <c:out value='${language}'/>
      </c:forEach>

      <p>
```

Listing 2.14 *param.jsp (cont.)*

```
  <%-- Show the value of the param.languages map entry,
            which is the first value for the languages
            request parameter --%>
       <c:out value="${'${'}param.languages} = ${param.languages}"/>
    </body>
</html>
```

The preceding JSP page does four things of interest. First, it displays the `lastName` and `firstName` request parameters, using the `param` implicit object. Since we know that those request parameters represent textfields, we know that they are a single value, so the `param` implicit object fits the bill.

Second, the JSP page displays all of the request parameters and their values, using the `paramValues` implicit object and the <c:forEach> action.[12] We use the `paramValues` implicit object for this task since we know that the HTML `select` element supports multiple selection and so can produce multiple request parameter values of the same name.

Because the `paramValues` implicit object is a map, you can access its values directly if you know the keys, meaning the request parameter names. For example, the third point of interest in the preceding JSP page iterates over the array of strings representing selected languages—`paramValues.languages`. The selected languages are accessed through the `paramValues` map by use of the key `languages`.

To emphasize the difference between `param` and `paramValues`, the fourth point of interest is the value of the `param.languages` request parameter, which contains only the first language selected in the HTML `select` element. A <c:out> action uses the EL expression `${'${'}` to display the characters `${` and another EL expression—`${param.languages}`—to display the first value for the `languages` request parameter.

Accessing Request Headers

You can access request headers just as you can access request parameters, except that you use the `header` and `headerValues` implicit objects instead of `param` and `paramValues`.

Like the `param` and `paramValues` implicit objects, the `header` and `headerValues` implicit objects are maps, but their keys are request header names.

12. See "The <c:forEach> Action" on page 154 for more information about <c:forEach>.

The `header` map's values are the first value specified for a particular request header, whereas the `headerValues` map contains arrays of all the values specified for that request header.

Figure 2–6 shows a JSP page that uses the `header` implicit object to display all of the request headers and the first value defined for each of them.

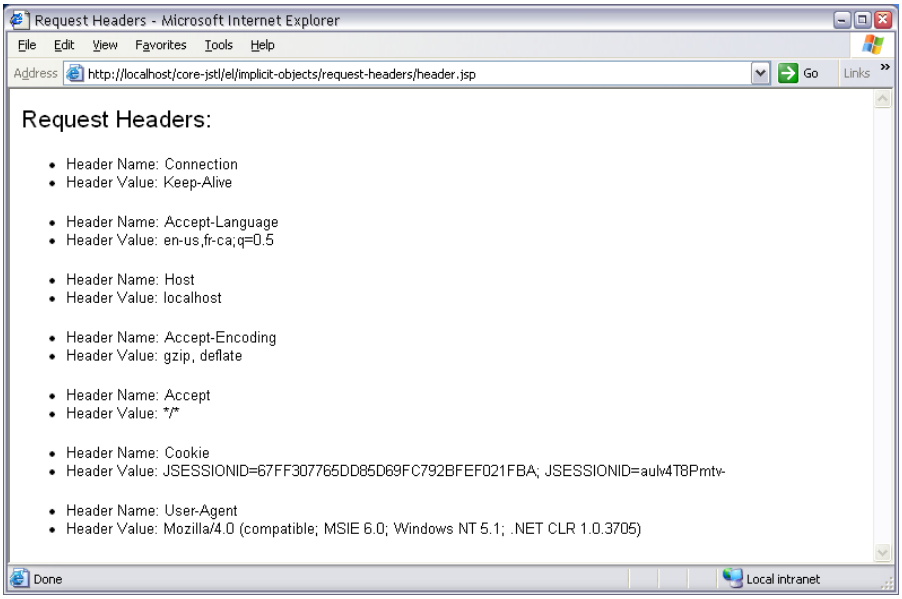

Figure 2–6 Accessing Request Headers with the `header` Implicit Object

The JSP page shown in Figure 2–6 is listed in Listing 2.15.

The keys stored in the `header` map are request header names and the corresponding values are strings representing request header values. You can also use the `headerValues` implicit object to iterate over request headers, like this:

```
<%-- Loop over the JSTL headerValues implicit object,
     which is a map --%>
<c:forEach items='${headerValues}' var='hv'>
   <ul>
      <%-- Display the key of the current item; that item
           is a Map.Entry --%>
      <li>Header name: <c:out value='${hv.key}'/></li>

      <%-- The value of the current item, which is
           accessed with the value method from
           Map.Entry, is an array of strings
```

Listing 2.15 *Accessing Requests Headers with the `header` Implicit Object*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
   <head>
      <title>Request Headers</title>
   </head>

   <body>
      <%@ taglib uri='http://java.sun.com/jstl/core' prefix='c' %>

      <font size='5'>
         Request Headers:
      </font><p>

      <%-- Loop over the JSTL header implicit object, which is a
           map --%>
      <c:forEach items='${header}' var='h'>
         <ul>
            <%-- Display the key of the current item, which
                 represents the request header name and the
                 current item's value, which represents the
                 header value --%>
            <li>Header Name: <c:out value='${h.key}'/></li>
            <li>Header Value: <c:out value='${h.value}'/></li>
         </ul>
      </c:forEach>
   </body>
</html>
```

```
           representing request header values, so
           we iterate over that array of strings --%>
      <c:forEach items='${hv.value}' var='value'>
           <li>Header Value: <c:out value='${value}'/></li>
      </c:forEach>
   </ul>
</c:forEach>
```

Unlike request parameters, request headers are rarely duplicated; instead, if multiple strings are specified for a single request header, browsers typically concatenate those strings separated by semicolons. Because of the sparsity of duplicated request headers, the `header` implicit object is usually preferred over `headerValues`.

Accessing Context Initialization Parameters

You can have only one value per context initialization parameter, so there's only one JSTL implicit object for accessing initialization parameters: `initParam`. Like the implicit objects for request parameters and headers, the `initParam` implicit object is a map. The map keys are context initialization parameter names and the corresponding values are the context initialization parameter values.

Figure 2–7 shows a JSP page that iterates over all the context initialization parameters and prints their values. That JSP page also accesses the parameters directly.

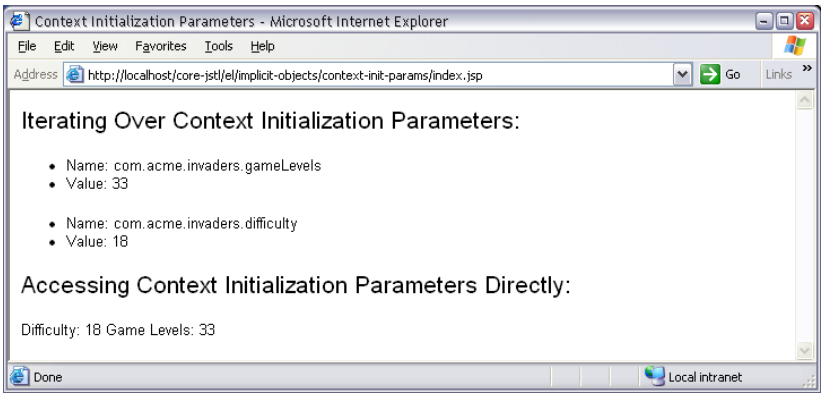

Figure 2–7 Accessing Initialization Parameters with the `initParam` Implicit Object

Before we discuss the listing for the JSP page shown in Figure 2–7, let's look at the deployment descriptor, listed in Listing 2.16, which defines two context initialization parameters: `com.acme.invaders.difficulty` and `com.acme.invaders.gameLevels`.

The context initialization parameters defined above are accessed by the JSP page shown in Figure 2–7 and listed in Listing 2.17.

The preceding JSP page uses the <c:forEach> action to iterate over the key/value pairs stored in the `initParam` map. The body of that action displays each key/value pair.

In the example discussed in "Accessing Request Parameters" on page 65, we accessed a request parameter by name like this: `${paramValues.languages}`. In the preceding JSP page, can we access an initialization parameter in a similar fashion with the `initParam` implicit object? The answer is yes, but in this case we have a problem because the initialization parameter name has `.` characters, which have special meaning to the expression language. If we try to access the `com.acme.invaders.difficulty` parameter like this: `${initParam.com.acme.invaders.difficulty}`, the expression

Listing 2.16 *WEB-INF/web.xml*

```
<?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/j2ee/dtds/web-app_2.3.dtd">

<web-app>
   <!-- Application-wide default values for the Acme Invaders
        online game -->
   <context-param>
      <param-name>com.acme.invaders.difficulty</param-name>
      <param-value>18</param-value>
   </context-param>

   <context-param>
      <param-name>com.acme.invaders.gameLevels</param-name>
      <param-value>33</param-value>
   </context-param>

   <welcome-file-list>
      <welcome-file>
         index.jsp
      </welcome-file>
   </welcome-file-list>
</web-app>
```

Listing 2.17 *Accessing Context Initialization Parameters*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
   <head>
      <title>Context Initialization Parameters</title>
   </head>

   <body>
      <%@ taglib uri='http://java.sun.com/jstl/core' prefix='c' %>

      <font size='5'>
         Iterating Over Context Initialization Parameters:
      </font><p>

<%-- Loop over the JSTL initParam implicit object,
           which is a map --%>
<c:forEach items='${initParam}' var='parameter'>
```

Listing 2.17 *Accessing Context Initialization Parameters (cont.)*

```
<ul>
            <%-- Display the key of the current item, which
                 corresponds to the name of the init param --%>
            <li>Name: <c:out value='${parameter.key}'/></li>

            <%-- Display the value of the current item, which
                 corresponds to the value of the init param --%>
            <li>Value: <c:out value='${parameter.value}'/></li>
          </ul>
      </c:forEach>

      <font size='5'>
         Accessing Context Initialization Parameters Directly:
      </font><p>

      Difficulty:
      <c:out value='${initParam["com.acme.invaders.difficulty"]}'/>

      Game Levels:
      <c:out value='${initParam["com.acme.invaders.gameLevels"]}'/>

   </body>
</html>
```

language will interpret that expression as an object's property named `difficulty`, which is not the interpretation we want.

The solution to this difficulty is to use the `[]` operator, which evaluates an expression and turns it into an identifier; for example, you can access the `com.acme.invaders.difficulty` initialization parameter like this: `${initParam["com.acme.invaders.difficulty"]}`. See "A Closer Look at the [] Operator" on page 56 for more information about the `[]` operator.

Accessing Cookies

It's not uncommon to read cookies in JSP pages, especially cookies that store user-interface-related preferences. The JSTL expression language lets you access cookies with the `cookie` implicit object. Like all JSTL implicit objects, the `cookie` implicit object is a map.[13] That map's keys represent cookie names, and the values are the cookies themselves.

Figure 2–8 shows a JSP page that reads cookie values, using the `cookie` implicit object.

13. The sole exception is the `pageContext` implicit object, which is not a map.

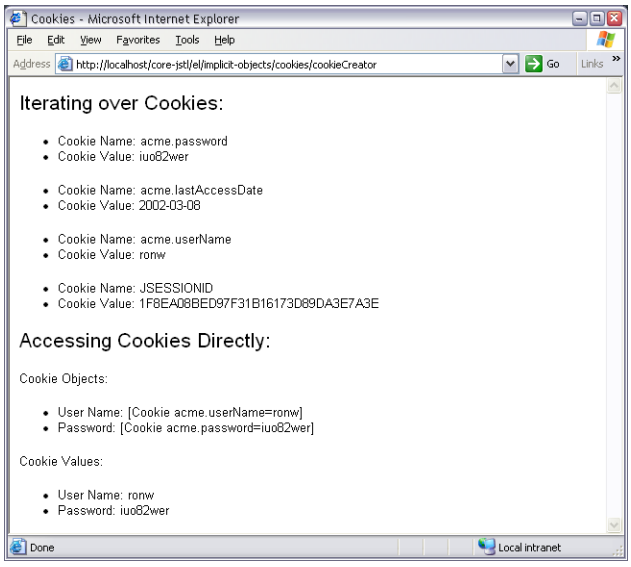

Figure 2–8 Accessing Cookies with the `cookie` Implicit Object

The JSP page shown in Figure 2–8 uses the `cookie` implicit object to iterate over all cookies and also accesses `Cookie` objects and their values directly. That JSP page is invoked with the URL `/cookieCreator`, which is mapped to a servlet that creates cookies. That servlet, after creating cookies, forwards to the JSP page shown in Figure 2–8. Listing 2.18 lists the Web application's deployment descriptor, which maps the URL `/cookieCreator` to the `CookieCreatorServlet` class.

Listing 2.18 *WEB-INF/web.xml*

```
<?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/j2ee/dtds/web-app_2.3.dtd">

<web-app>
   <servlet>
      <servlet-name>cookieCreator</servlet-name>
      <servlet-class>CookieCreatorServlet</servlet-class>
   </servlet>

   <servlet-mapping>
      <servlet-name>cookieCreator</servlet-name>
      <url-pattern>/cookieCreator</url-pattern>
   </servlet-mapping>
</web-app>
```

The `CookieCreatorServlet` class is listed in Listing 2.19.

Listing 2.19 *WEB-INF/classes/CookieCreatorServlet.java*

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

public class CookieCreatorServlet extends HttpServlet {
   public void doGet(HttpServletRequest request,
                     HttpServletResponse response)
                     throws IOException, ServletException {
      String[] cookieNames = {"acme.userName", "acme.password",
                              "acme.lastAccessDate"};
      String[] cookieValues = {"ronw", "iuo82wer", "2002-03-08"};

      // Create cookies and add them to the HTTP response
      for(int i=0; i < cookieNames.length; ++i) {
         Cookie cookie = new Cookie(cookieNames[i],
                                    cookieValues[i]);
         response.addCookie(cookie);
      }

      // Forward the request and response to cookies.jsp
      RequestDispatcher rd =
         request.getRequestDispatcher("cookies.jsp");
         rd.forward(request, response);
   }
}
```

The cookie creator servlet creates three cookies and adds them to the response before forwarding to `cookies.jsp`. That JSP page is listed in Listing 2.20.

Listing 2.20 *cookies.jsp*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
   <head>
      <title>Cookies</title>
   </head>

   <body>
      <%@ taglib uri='http://java.sun.com/jstl/core' prefix='c' %>

      <p><font size='5'>
         Iterating over Cookies:
      </font><p>
```

Listing 2.20 *cookies.jsp (cont.)*

```
<%-- Loop over the JSTL cookie implicit object, which is a
          map. If there are no cookies, the <c:forEach> action
          does nothing. --%>
      <c:forEach items='${cookie}' var='mapEntry'>
         <ul>
           <%-- The mapEntry's key references the cookie name --%>
            <li>Cookie Name: <c:out value='${mapEntry.key}'/></li>

            <%-- The mapEntry's value references the Cookie
                 object, so we show the cookie's value --%>
            <li>Cookie Value:
                <c:out value='${mapEntry.value.value}'/></li>
         </ul>
      </c:forEach>

      <p><font size='5'>
         Accessing Cookies Directly:
      </font><p>

      Cookie Objects:
      <ul>
         <li>
            User Name: <c:out value='${cookie["acme.userName"]}'/>
         </li>
         <li>
            Password:  <c:out value='${cookie["acme.password"]}'/>
         </li>
      </ul>

      Cookie Values:
      <ul>
         <li>
            User Name:
            <c:out value='${cookie["acme.userName"].value}'/>
         </li>
         <li>
            Password:
            <c:out value='${cookie["acme.password"].value}'/>
         </li>
      </ul>
   </body>
</html>
```

The preceding JSP page uses the <c:forEach> action to iterate over the entries contained in the `cookie` map. For each entry, the body of the <c:forEach> action displays the cookie's name and value. Notice that cookie values are accessed with the

expression `${mapEntry.value.value}`. The map entry's value is a cookie, which also has a `value` property.

The rest of the JSP page accesses cookie objects and their values directly. Because the cookie names contain `.` characters, they cannot be used as identifiers, so the preceding JSP page uses the `[]` operator to directly access cookies and their values.

Accessing Scoped Attributes

Since we started discussing JSTL implicit objects at "Implicit Objects" on page 64, we've seen how to access four types of objects:

- Request parameters
- Request headers
- Context initialization parameters
- Cookies

In addition to the specific types listed above, you can access any type of object that's stored in one of the four JSP scopes: page, request, session, or application. The expression language provides one implicit object for each scope:

- `pageScope`
- `requestScope`
- `sessionScope`
- `applicationScope`

Remember from our discussion in "Identifiers" on page 43 that identifiers refer to scoped variables; for example, the expression `${name}` refers to a scoped variable named `name`. That scoped variable can reside in page, request, session, or application scope. The expression language searches those scopes, in that order, for scoped variables.

The implicit objects listed above let you explicitly access variables stored in a specific scope; for example, if you know that the `name` scoped variable resides in session scope, the expression `${sessionScope.name}` is equivalent to `${name}`, but the latter unnecessarily searches the page and request scopes before finding the `name` scoped variable in session scope. Because of that unnecessary searching, `${sessionScope.name}` should be faster than `${name}`.

The scope implicit objects listed above—`pageScope`, `requestScope`, `sessionScope`, and `applicationScope`—are also handy if you need to iterate over attributes stored in a particular scope; for example, you might look for a

timestamp attribute in session scope. The scope implicit objects give you access to a map of attributes for a particular scope.

Figure 2–9 shows a Web application that displays all of the attributes from the scope of your choosing. The top picture in Figure 2–9 shows a JSP page that lets you select a scope, and the bottom picture shows a JSP page that lists the attributes for the selected scope.

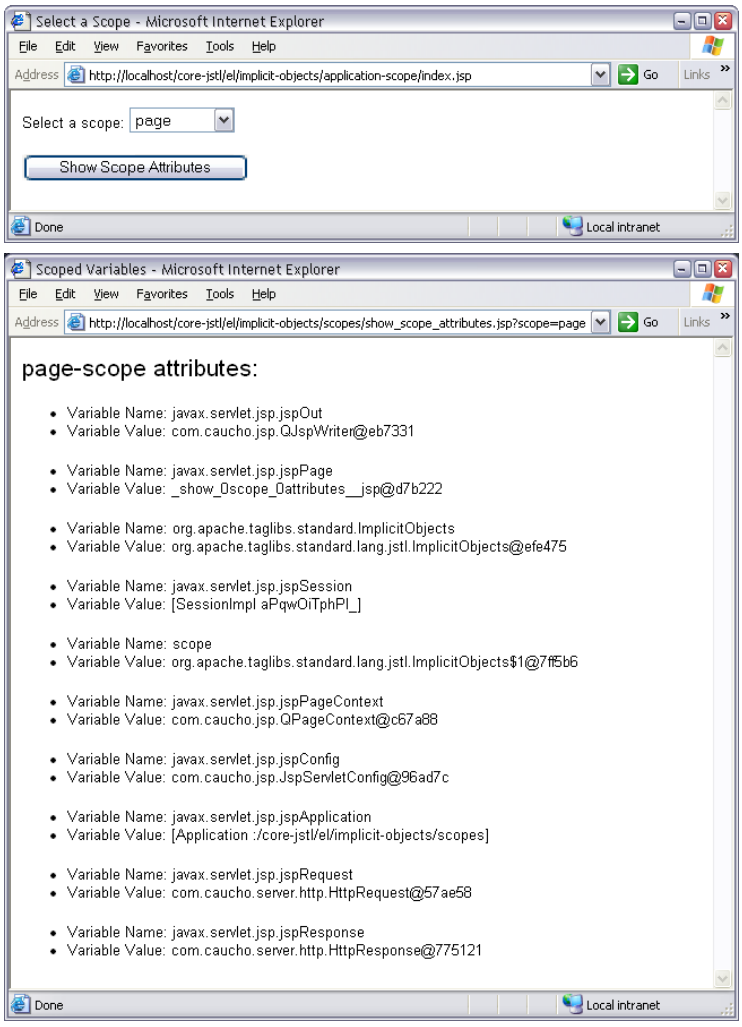

Figure 2–9 Accessing Scoped Variables for a Specific Scope with the `pageScope` Implicit Object

The JSP page shown in the top picture in Figure 2–9 is listed in Listing 2.21.

Listing 2.21 *Choosing a Scope*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
   <head>
      <title>Select a Scope</title>
   </head>

   <body>
      <form action='show_scope_attributes.jsp'>
         Select a scope:
         <select name='scope'>
            <option value='page'>page</option>
            <option value='request'>request</option>
            <option value='session'>session</option>
            <option value='application'>application</option>
         </select>

         <p><input type='submit' value='Show Scope Attributes'/>
      </form>
   </body>
</html>
```

The preceding JSP page creates an HTML form that lets you select a scope. That form's action is `show_scope_attributes.jsp`, which is listed in Listing 2.22.

The preceding JSP page is passed a request parameter named `scope` whose value is `"page"`, `"request"`, `"session"`, or `"application"`. The JSP page creates a page-scoped variable, also named `scope`, and sets it to the appropriate JSTL implicit object—`pageScope`, `requestScope`, `sessionScope`, or `applicationScope`—based on the `scope` request parameter. Then the JSP page loops over that implicit object and displays each scoped variable's name and value.

Accessing JSP Page and Servlet Properties

Now that we've seen how to access request parameters and headers, initialization parameters, cookies, and scoped variables, the JSTL implicit objects have one more feature to explore: accessing servlet and JSP properties, such as a request's protocol or server port, or the major and minor versions of the servlet API your container supports. You can find out that information and much more with the `pageContext` implicit object, which gives you access to the request, response, session, and application (also known as the servlet context). Useful properties for the `pageContext` implicit object are listed in Table 2.6.

Listing 2.22 *Showing Scoped Variables for a Specific Scope*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
   <head>
      <title>Scoped Variables</title>
   </head>

   <body>
      <%@ taglib uri='http://java.sun.com/jstl/core' prefix='c' %>

      <%-- Set a page-scoped attribute named scope to
           pageScope, requestScope, sessionScope, or
           applicationScope, depending on the value of a
           request parameter named scope --%>
      <c:choose>
         <c:when test='${param.scope == "page"}'>
            <c:set var='scope' value='${pageScope}'/>
         </c:when>
         <c:when test='${param.scope == "request"}'>
            <c:set var='scope' value='${requestScope}'/>
         </c:when>
         <c:when test='${param.scope == "session"}'>
            <c:set var='scope' value='${sessionScope}'/>
         </c:when>
         <c:when test='${param.scope == "application"}'>
            <c:set var='scope' value='${applicationScope}'/>
         </c:when>
      </c:choose>

      <font size='5'>
         <c:out value='${param.scope}'/>-scope attributes:
      </font><p>

      <%-- Loop over the JSTL implicit object, stored in the
           page-scoped attribute named scope that was set above.
           That implicit object is a map --%>
      <c:forEach items='${scope}' var='p'>
         <ul>
            <%-- Display the key of the current item, which
                 represents the parameter name --%>
            <li>Parameter Name: <c:out value='${p.key}'/></li>

            <%-- Display the value of the current item, which
                 represents the parameter value --%>
            <li>Parameter Value: <c:out value='${p.value}'/></li>
         </ul>
      </c:forEach>
   </body>
</html>
```

Table 2.6 `pageContext` Properties

Property	Type	Description
`request`	`ServletRequest`	The current request
`response`	`ServletResponse`	The current response
`servletConfig`	`ServletConfig`	The servlet configuration
`servletContext`	`ServletContext`	The servlet context (the application)
`session`	`HttpSession`	The current session

The `pageContext` properties listed in Table 2.6 give you access to a lot of information; for example, you can access a client's host name like this: `${pageContext.request.remoteHost}`, or you can access the session ID like this: `${pageContext.session.id}`.

The following four tables list useful request, response, session, and application properties, all of which are available through the `pageContext` implicit object.

Table 2.7 `pageContext.request` Properties

Property	Type	Description
`characterEncoding`	`String`	The character encoding for the request body
`contentType`	`String`	The MIME type of the request body
`locale`	`Locale`	The user's preferred locale
`locales`	`Enumeration`	The user's preferred locales
`new`	`boolean`	Evaluates to `true` if the server has created a session, but the client has not yet joined
`protocol`	`String`	The name and version of the protocol for the request; for example: `HTTP/1.1`
`remoteAddr`	`String`	The IP address of the client
`remoteHost`	`String`	The fully qualified host name of the client, or the IP address if the host name is undefined
`scheme`	`String`	The name of the scheme used for the current request; i.e.: `HTTP`, `HTTPS`, etc.
`serverName`	`String`	The host name of the server that received the request

Table 2.7 `pageContext.request` Properties *(cont.)*

Property	Type	Description
`serverPort`	`int`	The port number that the request was received on
`secure`	`boolean`	Indicates whether this was made on a secure channel such as `HTTPS`

Table 2.8 `pageContext.response` Properties

Property	Type	Description
`bufferSize`	`int`	The buffer size used for the response
`characterEncoding`	`String`	The character encoding used for the response body
`locale`	`Locale`	The locale assigned to the response
`committed`	`boolean`	Indicates whether the response has been committed

Table 2.9 `pageContext.session` Properties

Property	Type	Description
`creationTime`	`long`	The time the session was created (in milliseconds since January 1, 1970, GMT)
`id`	`String`	A unique session identifier
`lastAccessedTime`	`long`	The last time the session was accessed (in milliseconds since January 1, 1970, GMT)
`maxInactiveInterval`	`int`	The time duration for no activities, after which the session times out

Table 2.10 `pageContext.servletContext` Properties

Property	Type	Description
`majorVersion`	`int`	The major version of the Servlet API that the container supports
`minorVersion`	`int`	The minor version of the Servlet API that the container supports
`serverInfo`	`Set`	The name and version of the servlet container
`servletContextName`	`String`	The name of the Web application specified by the `display-name` attribute in the deployment descriptor

The JSP page shown in Figure 2–10 accesses some of the information available in the preceding tables: the request port, protocol, and locale; the response locale; the session ID and maximum inactive interval; and the servlet API version supported by the JSP container.

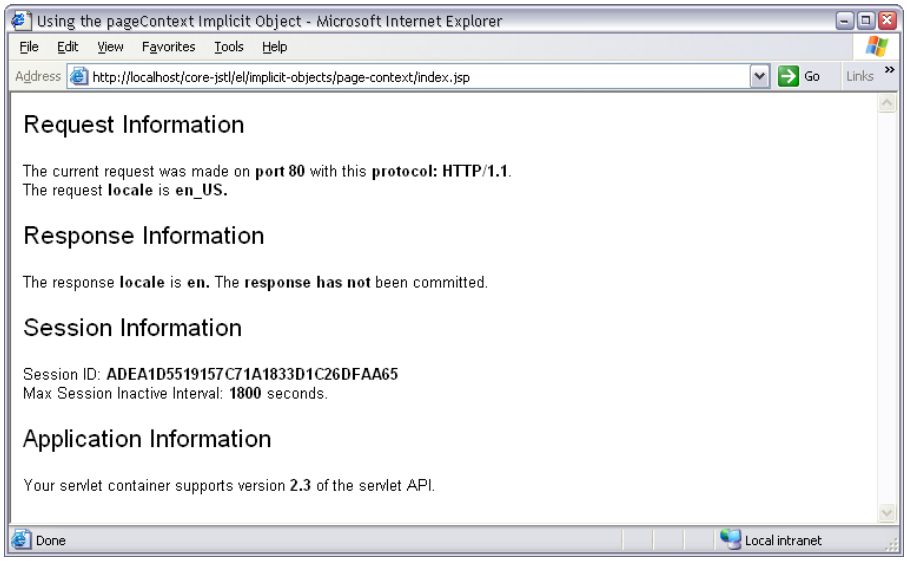

Figure 2–10 Using the `pageContext` Implicit Object

The JSP page shown in Figure 2–10 is listed in Listing 2.23.

Listing 2.23 *Accessing Servlet and JSP Properties*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
   <head>
      <title>Using the pageContext Implicit Object</title>
   </head>

   <body>
      <%@ taglib uri='http://java.sun.com/jstl/core' prefix='c' %>

      <%-- Show Request Information --%>
      <font size='5'>Request Information</font><p>

         <%-- Use the request object to show the server port and
                protocol --%>
The current request was made on <b>port
         <c:out value='${pageContext.request.serverPort}'/></b>
```

Listing 2.23 *Accessing Servlet and JSP Properties (cont.)*

```
with this <b>protocol:
        <c:out value='${pageContext.request.protocol}'/></b>.<br>

        <%-- Use the request object to show the user's preferred
             locale --%>
        The request <b>locale</b> is
        <b><c:out value='${pageContext.request.locale}'/>.</b>

        <p>

     <%-- Show Response Information --%>
     <font size='5'>Response Information</font><p>

        The response <b>locale</b> is
        <b><c:out value='${pageContext.response.locale}'/>.</b>

        <%-- Use the response object to show whether the response
             has been committed --%>
        The <b>response
        <c:choose>
           <c:when test='${pageContext.response.committed}'>
              has
           </c:when>

           <c:otherwise>
              has not
           </c:otherwise>
        </c:choose>
        </b> been committed.

        <p>

     <%-- Show Session Information --%>
     <font size='5'>Session Information</font><p>

        Session ID:
        <b><c:out value='${pageContext.session.id}'/></b><br>
        Max Session Inactive Interval:<b>
        <c:out
           value='${pageContext.session.maxInactiveInterval}'/>
        </b>seconds.

        <p>

     <%-- Show Application Information --%>
     <font size='5'>Application Information</font><p>
```

Listing 2.23 *Accessing Servlet and JSP Properties (cont.)*

```
<%-- Store the servlet context in a page-scoped variable
             named app for better readability --%>
        <c:set var='app' value='${pageContext.servletContext}'/>

        <%-- Use the application object to show the major and
             minor versions of the servlet API that the container
             supports --%>
        Your servlet container supports version<b>
        <c:out
           value='${app.majorVersion}.${app.minorVersion}'/></b>
        of the servlet API.
  </body>
</html>
```

The preceding JSP page accesses request, response, session, and application properties, using the `pageContext` implicit object. The end of that JSP page creates a page-scoped variable named `app` that references the servlet context (meaning the application). That page-scoped variable is subsequently used to access the Servlet API version supported by the JSP container. Sometimes it's convenient, for the sake of readability, to store a reference to one of the objects listed in Table 2.6 on page 82 in a page-scoped variable, as does the preceding JSP page.

2.8 Method Invocation

One of the most hotly debated topics within the JSTL expert group was whether the expression language should let you invoke arbitrary methods.

The major point of contention was whether that ability fit the philosophy of the expression language and whether it would encourage Java code in JSP pages. As you may have discerned so far and as you will learn more about as you explore JSTL actions throughout the rest of this book, the expression language and JSTL actions are implemented so that developers don't need to be concerned with types; for example, you iterate over a list, array, or comma-separated string in exactly the same fashion, without regard to their types, with the <c:forEach> action and EL expressions. If you could also invoke arbitrary methods on objects, that capability could compromise that intent and would open the door to another kind of expression language that contains EL expressions and Java statements.

The final decision for JSTL 1.0 was to *disallow direct method invocation* in the expression language.[14] You can only *indirectly* invoke a strict subset of methods for certain kinds of objects by specifying JavaBeans property names or array, list, or map indexes; see "A Closer Look at the [] Operator" on page 56 for more information.

Although that decision was probably for the best, you can still run into the need for method invocation pretty quickly; for example, consider the JSP page shown in Figure 2–11, which accesses the first item in a list.

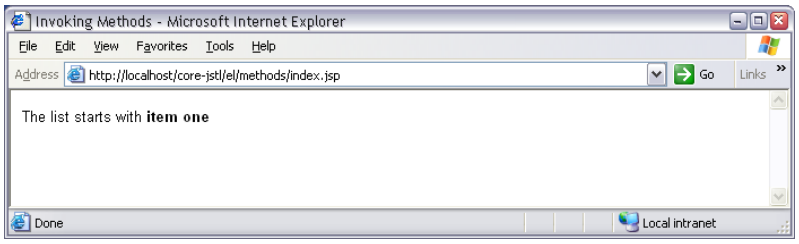

Figure 2–11 Accessing the First Item in a List

The JSP page shown in Figure 2–11 is listed in Listing 2.24.

Listing 2.24 *Accessing the First Item in a List*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
   <head>
      <title>Invoking Methods</title>
   </head>

   <body>
      <%@ taglib uri='http://java.sun.com/jstl/core' prefix='c' %>
      <%@ page import='java.util.LinkedList' %>

      <%
         LinkedList list = new LinkedList();
         list.add("item one");
         list.add("item two");
         list.add("item three");
         list.add("item four");
         list.add("item five");

         pageContext.setAttribute("list", list);
      %>

      The list starts with <b><c:out value='${list[0]}'/></b>
   </body>
</html>
```

14. An early draft of the JSP 2.0 specification includes direct method invocation for the expression language, but that feature may not make it into the final JSP 2.0 specification.

The preceding JSP page is simple: In a scriptlet, it creates a linked list and stores that list in page scope under the name `list`. Subsequently, the expression `${list[0]}` is used to access the first item in the list, and the output is `item one`.

So far, so good. But what if you want to access the last item in the list? To do that, you need to know how many items are in the list so that you can specify the proper position in the list. If you look at the Java documentation for the `LinkedList` class, you'll see that it has a `size` method that returns the number of items in the list. You might try to access the last item in the list like this:

```
<%-- Beware! this code will throw an exception --%>

The list starts with <b><c:out value='${list[0]}'/></b>
and ends with <b><c:out value='${list[list.size-1]}'/></b>
```

As you might guess, the preceding code fragment will throw an exception like the one shown in Figure 2–12.

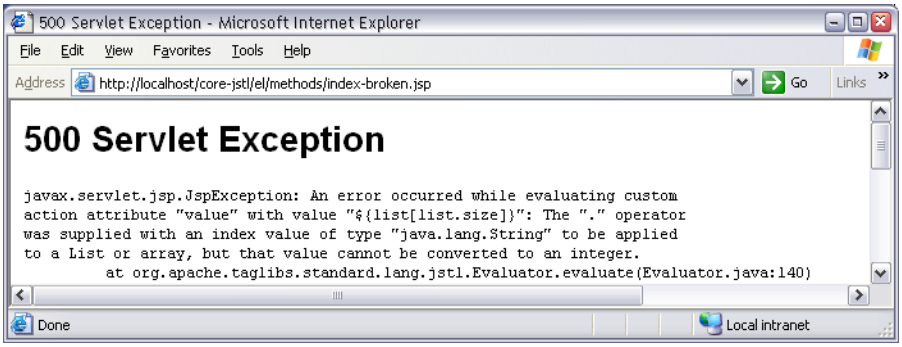

Figure 2–12 Trying to Access the Last Item in a List

The problem is that we are trying to invoke the list's `size` method (which is a valid `LinkedList` method), but it's *not a JavaBeans-compliant getter method*, so the expression `list.size-1` *cannot be evaluated*.

There are two ways to address this dilemma. First, you can use the RT Core library, like this:

```
<c_rt:out value='<%= list[list.size()-1] %>'/>
```

Second, if you want to avoid Java code in your JSP pages, you can implement a simple wrapper class that contains a list and provides access to the list's `size` property with a JavaBeans-compliant getter method. That bean is listed in Listing 2.25.

The preceding wrapper class has two JavaBeans properties: `list` and `size`; the former provides access to the list, and the latter provides access to the list's size. Listing 2.26 lists a JSP page that uses one of those wrappers.

Listing 2.25 *WEB-INF/classes/beans/ListWrapper.java*

```
package beans;

import java.util.List;

public class ListWrapper {
   private List list;

   // JavaBean accessors for first name
   public ListWrapper(List list) {
      this.list = list;
   }
   public List getList() {
      return list;
   }
   public int getSize() {
      return list.size();
   }
}
```

Listing 2.26 *Using a Wrapper to Access an Object's Properties*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
   <head>
      <title>Invoking Methods</title>
   </head>

   <body>
      <%@ taglib uri='http://java.sun.com/jstl/core' prefix='c' %>
      <%@ page import='java.util.LinkedList' %>
      <%@ page import='beans.ListWrapper' %>

      <%
         LinkedList list = new LinkedList();
         list.add("item one");
         list.add("item two");
         list.add("item three");
         list.add("item four");
         list.add("item five");

         ListWrapper listWrapper = new ListWrapper(list);
         pageContext.setAttribute("listWrapper", listWrapper);
      %>

      The first item is
      <b><c:out value='${listWrapper.list[0]}'/></b>
```

Listing 2.26 *Using a Wrapper to Access an Object's Properties (cont.)*

```
      and the last item is
      <b>
         <c:out value='${listWrapper.list[listWrapper.size-1]}'/>
      </b>

      <p>

      Here are all the items in the list:

      <p>

      <ul>
         <c:forEach var='item' items='${listWrapper.list}'>
            <li><c:out value='${item}'/></li>
         </c:forEach>
      </ul>
   </body>
</html>
```

Like the JSP page listed in Listing 2.24 on page 87, the preceding JSP page creates a list and populates it. But this time, the list is stored in a wrapper and the wrapper is stored in page scope. The JSP page accesses the list with the expression `listWrapper.list` and accesses the list's size with the expression `listWrapper.size`.

The JSP page listed in Listing 2.26 is shown in Figure 2–13.

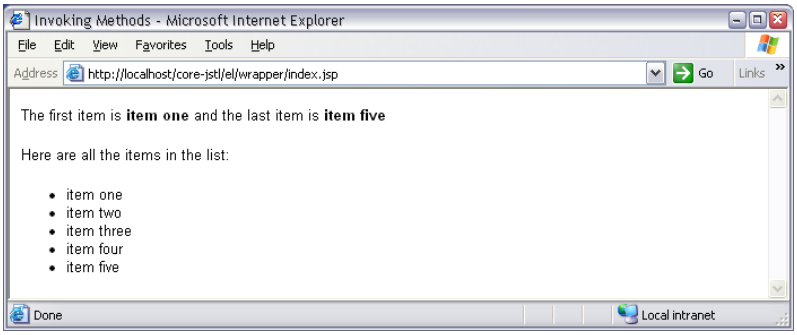

Figure 2–13 Using a JavaBeans Wrapper to Access a List's Size

The JSP page shown in Figure 2–13 and listed in Listing 2.26 displays the first and last items in the list and iterates over all of the items in the list. See "Iteration Actions" on page 150 for more information about iterating over collections.

2.9 EL Expressions in Custom Actions

The JSTL expression language is one of JSTL's most exciting features. If you implement JSP custom actions, you may be wondering how you can use the expression language for your own action attributes.

You can incorporate the expression language into your custom actions, but for JSTL 1.0, you cannot do it portably. Here's why: The JSP expert group is ultimately responsible for the expression language, which will be incorporated into JSP 2.0. When JSTL 1.0 was finalized—well before JSP 2.0—the JSP expert group had not yet defined a portable API for accessing the expression language. Because of that scheduling mismatch, until JSP 2.0 you will have to make do with writing code specific to the JSTL Reference Implementation.[15] JSP 2.0 will define a portable mechanism for accessing the expression language.[16]

This section shows you how to implement a custom action that permits EL expressions for an attribute using the JSTL 1.0 Reference Implementation.

Core Warning

For JSTL 1.0, it's not possible to use the EL for custom action attributes in a portable fashion.

Figure 2–14 shows a JSP page that uses a custom action to display values contained in a map. The maps shown in Figure 2–14 are accessed through some of the JSTL implicit objects discussed in "Implicit Objects" on page 64.

15. As this book went to press, negotiations were underway to put the expression language implementation of the JSTL Reference Implementation in Jakarta Commons.
16. See `http://java.sun.com/products/jsp/` to download the JSP 2.0 specification.

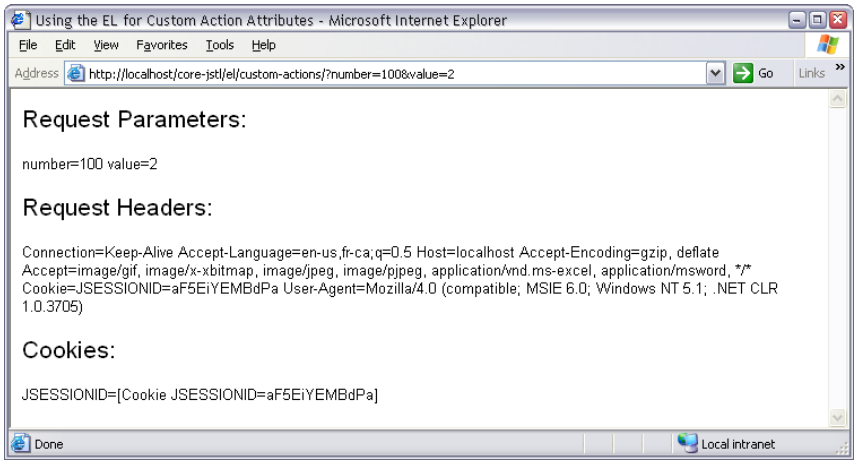

Figure 2–14 A Custom Action That Processes EL Expressions for Its Attribute

The JSP page shown in Figure 2–14 is listed in Listing 2.27.

Listing 2.27 *index.jsp*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
   <head>
      <title>Using the EL for Custom Action Attributes</title>
   </head>
   <body>
      <%@ taglib uri='WEB-INF/core-jstl.tld' prefix='core-jstl' %>

      <font size='5'>Request Parameters:</font>
      <p><core-jstl:showMap map='${param}'/>

      <p><font size='5'>Request Headers:</font>
      <p><core-jstl:showMap map='${header}'/>

      <p><font size='5'>Cookies:</font>
      <p><core-jstl:showMap map='${cookie}'/>
   </body>
</html>
```

The preceding JSP page uses a custom action—<core-jstl:showMap>—that displays values stored in a map. That custom action is unspectacular except for one feature: you can use the expression language to specify the action's `map` attribute.

Let's see how that custom action is implemented. First, we must specify a tag library descriptor (TLD) that defines the library and its lone action. That TLD, specified in `WEB-INF/core-jstl.tld`, is listed in Listing 2.28.

Listing 2.28 *WEB-INF/core-jstl.tld*

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
  PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
  "http://java.sun.com/dtds/web-jsptaglibrary_1_2.dtd">

<taglib>
   <tlib-version>1.0</tlib-version>
   <jsp-version>1.2</jsp-version>
   <short-name>JSTL Examples</short-name>
   <description>
      A custom action that shows how to incorporate the JSTL
      expression language for custom action attributes
   </description>

   <tag>
      <name>showMap</name>
      <tag-class>tags.ShowMapAction</tag-class>
      <body-content>JSP</body-content>
      <description>
         This action shows the values stored in a map
      </description>

      <attribute>
        <name>map</name>
        <required>true</required>
        <rtexprvalue>true</rtexprvalue>
       </attribute>
   </tag>
</taglib>
```

The preceding TLD specifies the name of the action—`showMap`—and the action's one required attribute, named `map`. The TLD also specifies the action's tag handler: `tags.ShowMapAction`, which is listed in Listing 2.29.

The preceding tag handler for the <core-jstl:showMap> action uses the Apache expression evaluator manager to evaluate the value specified for the `map` attribute with the `setMap` method. You pass the `ExpressionEvaluatorManager.evaluate` method the attribute's name, the expression specified for that attribute, the type that you expect the attribute to be, a reference to the tag handler and its page context. That method evaluates the expression and returns the appropriate object.

Listing 2.29 *WEB-INF/classes/tags/ShowMapAction.java*

```
package tags;

import java.util.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

// WARNING: non-standard class
import org.apache.taglibs.standard.lang.support.Expression
EvaluatorManager;

public class ShowMapAction extends TagSupport {
   private String mapName;
   private Map map;

   public void setMap(String mapName) {
      this.mapName = mapName;
   }
   public int doStartTag() throws JspException {
      // EL expressions must be evaluated in doStartTag()
      // and not in attribute setter methods, because servlet
      // containers can reuse tags, and if an attribute takes a
      // string literal, the setter method might not be called
      // every time the tag is encountered.
      map = (Map)ExpressionEvaluatorManager.evaluate(
               "map",                // attribute name
               mapName,              // expression
               java.util.Map.class,  // expected type
               this,                 // this tag handler
               pageContext);         // the page context

      if(map == null)
         return SKIP_BODY;

      Iterator it = map.keySet().iterator();
      JspWriter out = pageContext.getOut();

      while(it.hasNext()) {
         Object key = it.next(), value = map.get(key);

         try {
            if(value instanceof String[]) {
               String[] strings = (String[])value;

               for(int i=0; i < strings.length; ++i) {
                  out.println(strings[i]);
```

Listing 2.29 *WEB-INF/classes/tags/ShowMapAction.java (cont.)*

```
  }
            }
            else {
               out.println(key + "=" + value);
            }
         }
         catch(java.io.IOException ex) {
            throw new JspException(ex);
         }
      }
      return SKIP_BODY;
   }
}
```

2.10 Common Mistakes

All languages have their idiosyncrasies and pitfalls to avoid, and the JSTL expression language is no different. This section discusses some common mistakes that you are apt to make repeatedly. Once you are aware of them, it's easier to avoid them. Here are five of the most common JSTL expression language mistakes:

1. Forgetting curly braces
2. Forgetting `taglib` declarations
3. Neglecting to store variables in a scope
4. Using illegal characters for attribute names
5. Inadvertently using implicit objects

Forgetting Curly Braces

When you first start using the expression language, it can take awhile to remember to use the dollar sign and the *curly braces* for your expressions. Just as important, it can take awhile for expressions to look odd when the curly braces have been omitted. Here's a classic example:

```
<c:if test='counter.count == 1'>
   <%-- Do something the first time... --%>
</c:if>
```

The expression specified for the <c:if> test attribute will always evaluate to `false` because the value is a string that does not equal `"true"`.[17] Instead, you need to do this for the comparison to work:

```
<c:if test='${counter.count == 1}'>
   <%-- Do something the first time... --%>
</c:if>
```

Forgetting `taglib` Declarations

Even if you haven't yet read the Iteration Actions chapter in this book, you probably have a good idea what the following code fragment does:

```
<c:forEach var='item' begin='1' end='10'>
   <c:out value='${item}'/>
</c:forEach>
```

At first glance, it looks as though the preceding code fragment will print values from `1` to `10`, inclusive; however, that's not necessarily the case. If you *forget the `taglib` directive* for the JSTL core actions, the preceding code fragment will do nothing.

To make sure that the preceding code works as you expect, you need to remember the `taglib` directive, like this:

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

<c:forEach var='item' begin='1' end='10'>
   <c:out value='${item}'/>
</c:forEach>
```

Neglecting to Store Variables in a Scope

Although it's not recommended for production code, it is not uncommon for developers to create some temporary objects in a scriptlet that act as placeholders for data that will eventually come from another source; for example, you can create a hash map in a scriptlet that you can subsequently access with an EL expression, like this:

```
<%
   java.util.HashMap map = new java.util.HashMap();
   map.put("key One", "value One");
```

17. That criterion is from the Java documentation for `Boolean.valueOf(String)`. See Table 2.3 on page 62 for more information about expression language type coercions.

```
   map.put("key Two", "value Two");
   map.put("key Three", "value Three");
   map.put("key Four", "value Four");
   map.put("key Five", "value Five");
%>

<c:out value='${map["key One"]}'/>
```

You may think that the preceding code fragment will display the value of the first entry added to the map, but in actuality, it will display nothing at all because *the map created in the scriptlet was never stored in one of the JSP scopes*.

Once the map is placed in one of the JSP scopes, it can be accessed with an EL expression. Here is the corrected code fragment:

```
<%
   java.util.HashMap map = new java.util.HashMap();
   map.put("key One", "value One");
   map.put("key Two", "value Two");
   map.put("key Three", "value Three");
   map.put("key Four", "value Four");
   map.put("key Five", "value Five");

   pageContext.setAttribute("map", map);
%>

<c:out value='${map["key One"]}'/>
```

You can iterate over the items stored in the map created above like this:

```
<c:forEach var='item' items='${map}'>
   <c:out value='Key=${item.key}, Value=${item.value}'/>
</c:forEach>
```

Using Illegal Characters for Attribute Values

The preceding code fragment will print the key and value for each entry in a map. The following code, however, will not do the same:

```
<%-- The name an-item is not legal, so this produces no output --%>
<c:forEach var='an-item' items='${map}'>
   <c:out value='Key=${an-item.key}, Value=${an-item.value}'/>
</c:forEach>
```

The preceding code fragment will not produce any output because the name chosen for the scoped variable created by <c:forEach> is not a valid Java identifier—because it contains a dash—and therefore the preceding code will fail silently.

Inadvertently Using Implicit Objects

One final word of caution. Be careful that you don't inadvertently use the names of the JSTL implicit objects; for example, the following code fragment displays all of the request parameters, similar to the example discussed in "Accessing Request Parameters" on page 65:

```
<html>
   <head>
      ...
   </head>

   <body>
      <%@ taglib uri='http://java.sun.com/jstl/core' prefix='c' %>
      ...
      <%-- For every String[] item of paramValues... --%>
      <c:forEach var='parameter' items='${paramValues}'>
         <ul>
            <%-- Show the key, which is the request parameter
                 name --%>
            <li><b><c:out value='${parameter.key}'/></b>:</li>
            <%-- Iterate over the values -- a String[] --
                 associated with this request parameter --%>
            <c:forEach var='value' items='${parameter.value}'>
               <%-- Show the String value --%>
               <c:out value='${value}'/>
            </c:forEach>
         </ul>
      </c:forEach>
      ...
   </body>
</html>
```

The preceding code fragment works as advertised, but if you make this seemingly innocuous change—

```
...
<c:forEach var='param' items='${paramValues}'>
   <ul>
      ...
      <li><b><c:out value='${param.key}'/></b>:</li>
      ...
```

```
      <c:forEach var='value' items='${param.value}'>
         ...
      </c:forEach>
   </ul>
</c:forEach>
...
```

—the preceding code fragment will not work like the previous code fragment because `param` is an implicit object, not the current object of the iteration.