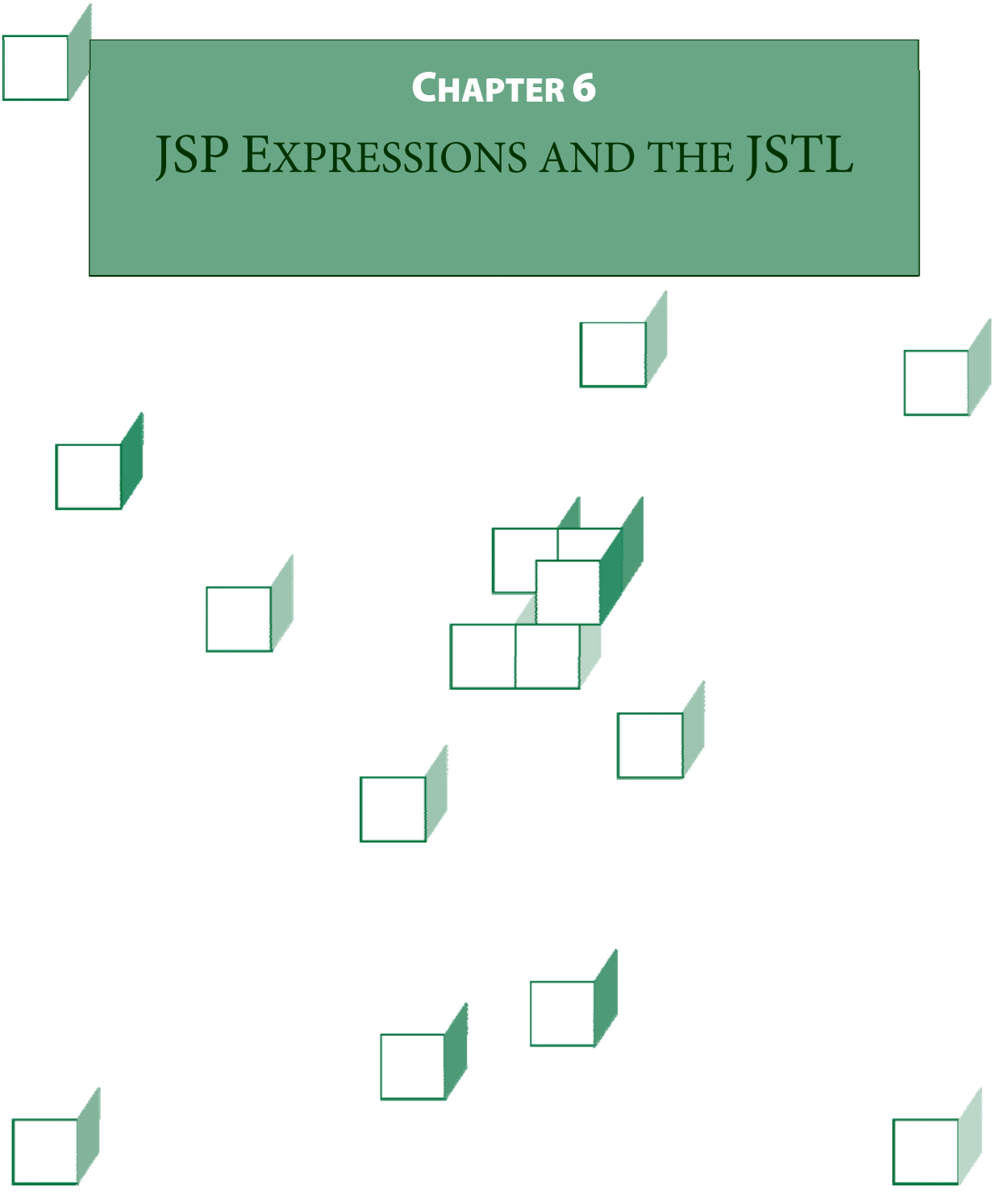
Chapter 6

JSP Expressions and the JSTL



OBJECTIVES

After completing "JSP Expressions and the JSTL," you will be able to:

- Describe the use of the JSP expression language in writing simple expressions to produce page content.
- Write JSP expressions and implement JSPs that use them in favor of script expressions and scriptlets.
- Describe the role of the JSP Standard Tag Library in JSP development.
- Implement JSPs that use basic JSTL actions to simplify presentation logic.
- Compare script-driven authoring with scriptless authoring that uses the EL and JSTL, and make well-informed decisions as to which style to employ in your own JSP development.

Going Scriptless

- **As we discussed in the previous chapter, it's best in large-scale application design to minimize the amount of business logic in front-line JSPs.**
- **JavaBeans and standard actions are a good start, but (as we also discussed) the standard action syntax is unwieldy, especially when performing a simple chore such as getting a bean property value.**
- **The JSP 2.0 expression language, or EL, offers a minimal syntax for reading information out of an object or graph of objects.**
  - The EL realizes the vision of dynamic page development in which page authors can quickly and simply invoke application behaviors and derive necessary values for pages, without letting a lot of Java code creep onto the page.
  - JSP expressions do what script expressions were meant to do, but since they don't rely on Java, they are much simpler, and in their own way more powerful.
- **In support of JSP expressions, the JSP Standard Tag Library, or JSTL, provides some simple custom actions for basic processing logic and formatting.**
  - JSTL tags (or perhaps "**JSTs**") can replace much of what JSP 1.2 developers had to do in scriptlets.

The JSP Expression Language

- **In fact, the expression language was first conceived in support of the JSTL, rather than the other way around.**
  - EL can be used in attribute values of JSTs.
  - The JSTL specification lays out a prototypical EL, but cedes the ultimate responsibility for its grammar to JSP.
  - EL can be used on its own in a page supported by a JSP 2.0 container – the container will translate it at request time.
  - When used as an argument to a JST, the expression can be evaluated by the tag handler, or the container can do it – depending mostly on the container version.

Using JSP Expressions

- JSP expressions are always of the general form **`${expression}`**, and can be embedded directly in JSP template content:

```
<p>The value of myBean's name property is
  ${myBean.name}.</p>
```

- They can be passed to a JST in attribute values:

```
<fmt:formatDate value="${thatJob.completion}"
  ... />
```

  - In both cases, expressions can be intermingled with static text, and multiple expressions can appear in sequence.

```
<c:set var="x"
       value="Signal ${n} level: ${level}dB" />
```

EL Syntax

- **The expression language is so simple to use that it's a little tricky to explain!**
  - Most of the syntax is highly intuitive.
  - EL borrows elements from ECMAScript and XPath.
  - It will also look familiar to Java programmers, although there are some important differences in how certain operators are evaluated.
- **Remember that the EL is "native" to JSP, and although it will be interpreted and translated to Java code in practice, it is not specifically bound to Java or to any other programming language.**
- **We'll look at some of the major grammatical elements on the next few pages:**
  - Type model
  - Literal values
  - Variables
  - Arithmetic operators
  - Indexing/member operators

Type Model and Literal Values

- **The JSP expression language supports the following types – these are not Java types per se:**
  - **Boolean**
  - **Integer**
  - **Floating-point number**
  - **String**
  - **Object**
- **Ultimately, though, everything in EL is an object.**
  - There are no "primitive" types.
- **Literal values can be stated for all of these types:**
  - "true" and "false" are **boolean** literals.
  - **Integer** literals are recognized in tokens that start with a digit and don't conclude with or embed a decimal point; a leading plus or minus sign is also legal.
  - **Floating-point** literals are like integer literals but can include a decimal point and exponentials ("5.4e-09", e.g.).
  - Any **quoted string** of characters can form a string literal. Either single or double quotes may be used, and one can be successfully embedded in the other. Either can also be escaped, as in "He said, \"Hello!\"".
  - The token **null** is a literal indicating the absence of a value.

Variables

- **The evaluator will treat anything it can't recognize as being a literal or an operator as a variable.**
- **Variables are referenced by name in expressions.**
  - The EL's **variable** is what script elements know as an **object reference**, and an expression will always be referring to objects in the same four web scopes we've seen in earlier chapters: page, request, session, and application.
  - Page scope is assumed when evaluating an expression.
  - Other scopes can be derived using implicit objects – more on this in a moment.
  - There is no question of defining a scope for a new object, since the EL cannot create objects.

Indexing/Member Operators

- **To read a property of an object, use the "." operator.**
  - This is a lot like the **<jsp:getProperty>** action, where the variable before the dot is the bean and the token after the dot is a property name.
  - That is, the following constructs are conceptually equivalent:

```
${A.B}
<jsp:getProperty name="A" property="B" />
```

- **To read an indexed property, use the "[]" operator.**
  - Standard actions can't do this!
  - The expression before the brackets must be a collection, or must be an indexed property on another object.
  - The expression in the brackets might be a numeric index into the collection or property, or might be any other type – the container will simply pass the value to the appropriate accessor method.
  - There are two primary usages here:

```
${myCollection[5]}
${myBean.myIndexedProperty[3]}
```

  - Note in the second case that the expression **myBean.myIndexedProperty** would not be legal by itself.

Indexing/Member Operators

- **Either the dot or bracket operators can be chained together, and nested.**

```
${yourBean.location.path}
${this["that"].member.collection[4]}
${param[preferences.keyParameter]}
```

- **There's also an overlap in what you can do with dot and bracket operators.**
  - To wit, the following two expressions are equivalent, for any object reference **A** and any property name **B**:

```
${A.B}
${A["B"]}
```

  - This follows a lead from ECMAScript.
  - Note however that only the bracket operators can get a property name dynamically, as in:

```
${A[somePropertyName]}
```

  - ... and there is no dot syntax that does this.

Arithmetic and Logical Operators

- **EL supports basic arithmetic and logical operators, much like Java, ECMA, and XPath.**
  - In fact, its operator set is very nearly the superset of operators from those languages.
  - Note the redundancies in the following lists.
  - Arithmetic operators are:

```
+ - * / div % mod
```

  - Logical operators are:

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

- **Operator precedence is about as expected; consult the JSP 2.0 specification for the exact order.**
- **Parentheses may be used to promote an expression to be evaluated before another.**

```
${x * (y + z)}
```

The empty Operator

- **Since EL does not provide for method invocation, only for property reads, it needs a specific syntax to check the size of a collection.**
  - If the collection type supports a property such as **count** or **size**, then that can easily be called.
  - However, the Java collection classes (such as Vector) do not uniformly support any such property, and the EL doesn't provide a means of calling the **size** method.
- **What EL does provide is a means of testing for emptiness of a collection: this is the `empty` operator.**
  - This is a unary prefix operator, so **`empty A`** evaluates to true if A is either **null** or a collection with no elements.

EL Functions

- **The expression language also supports a function-calling syntax.**
- **This is not object-oriented, as one might first expect.**
- **It supports calling C-style functions in a flat API, rather than member functions on a specific object.**
- **Functions are implemented as static methods on Java classes which are packed into tag libraries.**
  - We'll look more closely at tag libraries in the following chapter. To invoke a function in a given tag library, import the library with a **<%@ taglib %>** directive.
  - Invoke the function by writing its qualified name followed by a parenthesized list of arguments, separated by commas.

```
<%@ taglib prefix="my" uri="someURI" %>
...
${my:formatOutput (part1, part2)}
<c:set var="value"
       value="${my:getSomeValue (5, x, y)}" />
```

Type Coercion

- **EL is a weakly typed language.**
  - That is, there is no declaration of the type of any expression, variable, sub-expression, etc.
  - There is no explicit type-casting, as there is in Java.
- **Thus EL evaluation works from the outside in.**
- **In this way the evaluator can determine the expected type of an expression before evaluating it.**
- **The expression's value will then be coerced to the appropriate type.**
  - This can be as simple as a number being converted to its string representation, or vice-versa.
  - Not every value can be successfully coerced to every type.
- **For the most part, type coercion is transparent to the expression author, and a great deal of casting and conversion code is quietly left at the roadside.**

Error Handling

- **Part of the philosophy behind the EL is that it is to be used in presentation code, and that in that context it's typically best to produce an incomplete result than to fail completely on a recoverable error or warning.**
- **When evaluation of an expression or sub-expression fails, a default value will be provided by the evaluator for the required type, and expression evaluation will continue from there.**
  - Default values are empty strings and zero values.
  - Thus a common result while an expression is still in development and debugging is no output at all.
- **This can make EL tricky to debug: sometimes we can't tell the difference between an expression that just didn't turn up a result and one that's mistyped!**
- **You will see "hard failures" in the form of exceptions in at least two other cases:**
  - **Syntax errors** in your expression (that is, not just the wrong name for a bean property, but basic syntax problems such as `${A[B}`)
  - **Exceptions thrown from Java code** that might be invoked as part of evaluating the expression

Implicit Objects

- **As with script elements, the EL relies on a set of implicit objects provided by the JSP container.**
- **The set is not exactly the same as that for scripting elements.**
- **The full set is shown on the following page.**
  - Like the scripting object set, there are **redundancies** – objects that could be derived from other implicit objects.
  - In fact, there is probably a higher degree of redundancy in the EL set – and a correspondingly higher level of **convenience** in coding.
  - For example, **headers**, **cookies**, and **initialization parameters** are all available as simple maps, which are easy to dereference using dot/bracket notation, as in **`${cookie["customerName"]}`**.

List of Implicit Objects

- Implicit objects for JSP expressions:

Name	Description
pageContext	The PageContext object
pageScope	Map of page-scope object references
requestScope	Map of request-scope object references
sessionScope	Map of session-scope object references
applicationScope	Map of application-scope object references
param	Map of request parameters – name to single value
paramValues	Map of request parameters – name to array of all values
header	Map of headers – name to single value
headerValues	Map of headers – name to array of all values
cookie	Map of pertinent cookies – name to Cookie
initParam	Map of initialization parameters – name to Object

Echo in EL

EXAMPLE

- In **Examples\Echo\EL\Echo.jsp**, the Echo application from earlier in the module has been rewritten to use a JSP expression:

```
<html>

<head>
  <title>Echo -- Using JSP Expressions</title>
</head>

<body bgcolor="#F8F8E8" >

  <center>
    <h2>Echo &#8212; Using JSP Expressions</h2>
  </center>

  <p><b>Welcome, ${param["name"]}!</b></p>

</body>

</html>
```

The JSP Standard Tag Library

- **Prior to the development of JSP 2.0, custom tag library development for JSPs was thriving and maturing.**
- **A number of libraries have emerged that have nearly universal applicability to JSP authoring tasks.**
- **Chief among these is a set that has been adopted and formalized under the JCP as the JSP Standard Tag Library, or JSTL.**
  - These actions are so common that it is recommended that JSP containers provide native support for them.
  - Until containers do so, JSTL implementations can be deployed on a per-application basis, just like any custom tag library.
  - In this module's exercises, we're using a JSTL implementation from Apache, extracted from the Jakarta Taglibs project.
- **As EL borrows heavily from XPath (along with ECMAScript), JSTL can be seen to be guided in large part by the design of XSLT.**

JSTL Namespaces

- **The JSTL consists of four distinct tag libraries, each with its own namespace (naturally), and each with a suggested prefix.**
  - The namespace is prescribed by the JSTL specification.
  - The prefix is ultimately up to the page author, as per the usual rules for JSP custom tag libraries. The page author chooses the prefix in the **taglib** directive.
- **The four tag libraries that make up the JSTL are:**

Name	Prefix	Namespace URI
Core	c:	http://java.sun.com/jsp/jstl/core
Formatting	fmt:	http://java.sun.com/jsp/jstl/fmt
SQL	sql:	http://java.sun.com/jsp/jstl/sql
XML	x:	http://java.sun.com/jsp/jstl/xml

Using JSTL in a Page

- **To bring any of the four JSTL libraries into play in a JSP, one needs to take a few steps to inform the container how to translate an action into the corresponding Java code.**
  - Any action will have a prefix and a name, and then any number of attributes.

```
<c:forEach var="x" items="${myCollection}" > ...
```

  - The page must include a **taglib** directive that maps prefix to URI.

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

  - This URI must be mapped by a **tag library descriptor**, or **TLD**, to various **tag handlers** for each tag in the library.
  - The TLD may be declared explicitly in **web.xml**, or it may be found automatically in the **META-INF** directory of any JAR file deployed in **WEB-INF\lib**.
  - The Java class must be available to the container at runtime, which typically means that a JAR including the class will be deployed with the application, under **WEB-INF\lib**.
- **The JSTL reference implementation is deployed as part of our web application.**
  - You can see the files **jstl.jar** and **standard.jar** under **Application/WEB-INF/lib**.

The Core Actions

- **We'll consider just a few of the JSTL actions in this chapter.**
- **Our purpose for the moment is not to learn the JSTL in depth, but rather:**
  - To learn its role in JSP authoring, especially as it can assist widespread use of the JSP expression language
  - To develop a general familiarity with its syntax and common usage, so as to be able to recognize it where it occurs
- **For these purposes, we'll study just two of the core actions**
  - **`<c:set>`**
  - **`<c:if>`**

Modifying Information with <c:set>

- **The core library includes one action in particular that's very helpful in conjunction with JSP expressions: the `<c:set>` action.**
- **Remember that EL cannot write values into objects; it can only read and produce them to the output stream.**
- **`<c:set>` provides the basic ability to declare page-scope variables, and to initialize and to modify their values.**

```
<c:set var="var-name"
  scope="scope" value="new-value" />
```

- **It is not too different from `<jsp:useBean>`, but:**
  - It cannot define the Java **class** of a new variable, as **`<jsp:useBean>`** can; the variable type can only be determined implicitly based on value expression.
  - It can **initialize** a variable to a value – something like **`<jsp:useBean>`** and **`<jsp:setProperty>`** in one step.
  - Some of the attributes of **`<c:set>`** can include JSP **expressions**.

Conditional Processing with <c:if>

- **Another convenient core action is `<c:if>`, which includes other JSP content (static or dynamic) and processes it only if a test expression evaluates to true.**

```
<c:if test="${param['a'] == param['b'] and
              param['a'] == param['c']}">
  <td>Sphere</td>
</c:if>
```

- **A more complex combination of three other core actions is required to get an if/else or switch/case structure.**
  - The actions are **`<c:choose>`**, **`<c:when>`**, and **`<c:otherwise>`**.

```
<c:choose>
  <c:when test="${x % 2 == 0}" >even</c:when>
  <c:otherwise>odd</c:otherwise>
</c:if>
```

  - We'll keep it simple in exercises in this chapter, and stick to **`<c:if>`**.
  - A crude but effective way to get if/else behavior is simply to write two **`<c:if>`** actions with opposite tests.

Odd and Even

LAB 6A

In this lab you will re-implement the Odd and Even application using JSP expressions and conditional logic based in JSTL tags.

Detailed instructions are contained in the Lab 6A write-up at the end of the chapter.

Suggested time: 30 minutes.

Using Beans with JSTL

- **The one major limitation of the core action set is that it offers no way to instantiate new objects.**
  - **`<c:set>`** can find an existing object at a given scope and initialize it.
  - It can declare a new variable at any scope, but there is no way to define the type of the variable.
  - Thus only common types such as strings and numbers can be instantiated; specific application-defined classes cannot be specified.
- **The philosophy behind this intentional omission is that other components, such as servlets, should be choosing the types of objects, and populating them.**
- **If a JSP needs to create objects, it can use `<jsp:useBean>`, and the resulting objects will be visible to EL and to JSTs.**
- **Another common behavior for which the JSTL does not provide is adding an element to a collection.**

The Formatting Actions

- **A second library under the JSTL provides for formatted output, localization and internationalization.**
  - **`<fmt:setLocale>`** sets a default locale for other formatting actions, at any chosen scope.
  - **`<fmt:setTimeZone>`** accomplishes the same thing for time zones.
  - Various actions in this library will format or parse dates and times under this locale and time zone. (Time zone can be overridden per action.)
  - For instance, **`<fmt:formatDate>`** will produce formatted date and/or time output according to a specified format or style.

```
<jsp:useBean id="now" class="java.util.Date" />
<fmt:formatDate value="${now}" />
```

Date and Time

LAB 6B

Optional

In this lab you will re-implement the Date and Time application using JSP expressions and core and formatting tags from the JSTL.

Detailed instructions are contained in the Lab 6B write-up at the end of the chapter.

Suggested time: 15-30 minutes.

Scripts vs. EL/JSTL

- **Let's compare scripting to scriptless authoring:**
  - Many common tasks and information items can be encoded far more easily using EL than in scriptlet code.
  - Still, scriptlets offer the full power of the Java language in the JSP.
- **For experienced Java coders, EL and JSTL can seem unwieldy at first, and may not appear to offer as much power.**
- **Java code does not really belong on front-line JSPs, however.**
  - The arguments against this have been discussed and have to do with reusability of business and presentation logic, and division of labor between page authors and programmers.
  - Where Java is truly necessary to implement a given body of logic, the argument is simply that the Java code should be transferred from the JSP itself to a component: either a JavaBean or a tag handler class.
- **Your work in Lab 6A should help to illustrate the advantages of scriptless authoring, even by the narrow criteria of simplifying logic on a single page.**
  - Additional JSTL actions offer even more benefit.
  - For instance there is a **`<c:forEach>`** that facilitates looping.
  - Further study of the JSTL is beyond the scope of this module.

In this lab you will re-implement some pages of the Electronic DJ application using EL, JSTL and standard actions to manage JavaBeans.

Detailed instructions are contained in the Lab 6C write-up at the end of the chapter.

Suggested time: 15 minutes.

SUMMARY

- **The JSP expression language is simple and intuitive to use, and can make many page-authoring tasks much easier.**
- **It is also independent of Java, which has advantages:**
  - Page authors don't need extensive Java skills.
  - JSP can conceivably be implemented in a non-Java context, and can be made to work with scripting languages other than Java, if scripting is desired at all.
- **Custom actions offer the greatest potential for extending the power of JSP for web presentation, by encapsulating business or presentation logic at any scope from a single application up to an entire industry.**
- **The JSTL is just the tip of an iceberg of custom tag libraries, many of which are available for free or otherwise easily procured.**
- **The JSP 2.0 authors "expect …the prevalent use of script-less pages."**
  - The aim of JSP 2.0 is to wean JSP authors off of scripting elements entirely, though it is clear in the JSP specification language that this is not expected to happen overnight.
  - The expression language and expanded use of standard and custom actions (including the JSTL) will facilitate this shift.