

Jeff Heaton

QUICK > CONCISE > PRACTICAL

JSTL:

JSP Standard Tag Library

SAMS

KICK START



JSTL: JSP Standard Tag Library Kick Start

By Jeff Heaton

Publisher : Sams Publishing
Pub Date : September 19, 2002
ISBN : 0-672-32450-4.htm
Pages : 432

The *JSP Standard Tag Library* is a collection of commonly used functions and tools invaluable to JSP developers to avoid re-creating the same functions on site after site. Sun has indicated that JSP development should be based around using tag libraries going forward, and will release JSP STL, as their official library. This book starts with an in-depth discussion of the JSP STL, then goes beyond the standard library to teach developers to create their own tags to further encapsulate the most common features of their specific applications. Along the way, readers will also learn to use tags to access data, process XML, handle expressions, and further customize pages for international visitors. Later chapters explain how readers can expand the Standard Tag Library by creating their own tags.

Table of Contents

Table of Contents	2
Copyright	5
Copyright © 2003 by Sams Publishing	5
Trademarks	5
Warning and Disclaimer	5
Credits	5
Dedication	6
About the Author	7
Acknowledgments.....	8
Tell Us What You Think!	9
Introduction.....	10
The Benefits of JSTL.....	10
JSTL and JSP	10
Who This Book Is For.....	11
Structure and Organization of the Book	11
The Forum Application.....	12
Source Code and Updates	12
Chapter 1. Understanding JSP Custom Tags	13
The Role of Tags in JSP.....	13
Introducing JSTL	18
Installing JSTL.....	20
Chapter 2. Programming the JSP Standard Tag Library.....	25
Understanding Scoped Variables	25
Accessing Application Data.....	35
The Basics of Web Application Programming	45
Summary	52
Chapter 3. Understanding Basic Tag Logic	53
Exception Processing	53
Using Conditionals.....	58
A Chat Application	65
Summary	69
Chapter 4. Using the Expression Language.....	70
JSTL Expression Tags	70
Using the EL Expression Language.....	81
Using the RT Expression Language	90
Summary	95
Chapter 5. Collections, Loops, and Iterators.....	96
Understanding Collections	96
The Iteration Tags	96
Using Iterators.....	100
Summary	116
Chapter 6. Formatting Data with Tags.....	117
Formatting.....	117
Parsing.....	131

Time Zones	141
Applying Date Formatting	145
Summary	151
Chapter 7. Accessing Data with SQL Tags	152
Introducing the Forum Example	152
Understanding JDBC Drivers	162
Using SQL Tags.....	167
Implementing the Forum Example	172
A General Query Engine.....	190
Summary	194
Chapter 8. Accessing Data with XML Tags	195
Understanding XML	195
Understanding XPath	197
Understanding XML Core Tags.....	198
Using XML Flow-Control Tags.....	207
Transforming XML with XSLT.....	216
Summary	223
Chapter 9. Accessing Internet Resources with JSTL	224
The URL-Related Tags	224
Understanding Internet Resources	237
Summary	243
Chapter 10. Understanding JSTL Internationalization	245
The I18N-Related Tags.....	245
Resource Bundles.....	253
A Multilingual Forum Application	262
Summary	268
Chapter 11. Creating Your Own Tag Libraries	269
Developing Custom Tag Libraries.....	269
The Components of a Tag Library.....	276
Summary	305
Chapter 12. Debugging and Deploying Tag Libraries.....	306
Debugging.....	306
Debugging with an IDE	312
Deploying Web Applications.....	321
Summary	324
Appendix A. JSTL Reference	325
The Core Tags.....	325
The I18N Tags	332
The Relational Database Tags (SQL)	339
The XML Tags.....	343
Appendix B. Installing JSTL and Tomcat	349
Installing JDK	349
Installing Tomcat	349
Installing the Book Examples	353
The Classpath and Search Path	354
Installing JSTL without the Examples.....	356

Appendix C. Installing MySQL.....	358
Obtaining and Installing MySQL.....	358
Setting Up MySQL	361
Creating the Forum Example.....	364
Installing a JDBC Driver	365
Appendix D. Unix Installation Notes.....	367

Copyright

Copyright © 2003 by Sams Publishing

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

Library of Congress Catalog Card Number: 2002106117

Printed in the United States of America

First Printing: September 2002

04 03 02 4 3 2 1

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The author(s) and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the CD or programs accompanying it.

Credits

Executive Editor

Michael Stephens

Acquisitions Editor

Todd Green

Managing Editor

Charlotte Clapp

Copyeditor

Elizabeth Welch

Compositor

Regina Rexrode

Proofreader

Nancy Sixsmith

Technical Editor

Avery Regier

Multimedia Developer

Dan Scherf

Team Coordinator

Lynn Williams

Interior Designer

Gary Adair

Cover Designer

Gary Adair

Dedication

This book is dedicated to my sister, Carrie Heaton, and my future brother-in-law, Jeremi Spear. I wish you both many years of happiness after your wedding on March 29, 2003.

About the Author

Jeff Heaton is a software designer for Reinsurance Group of America, Inc. (RGA). Jeff also teaches Java at St. Louis Community College at Meramec, and is a graduate student at Washington University in St. Louis. A Sun Certified Java Programmer specializing in Internet, socket-level/spidering, and artificial intelligence programming, Jeff is a member of the IEEE. He can be contacted by email at heatonj@heat-on.com or through his Web site at <http://www.jeffheaton.com/>.

Acknowledgments

First, I'd like to thank everyone at Sams Publishing who made this process productive and fun. Thanks go to my acquisitions editor, Todd Green, for believing in this project and helping to get it started. I'd also like to thank my editor, Elizabeth Welch, for helping to make the text easy to read and follow.

In addition, I want to thank Jenny Yao for providing the translation of our forum application to Chinese. And I am indebted to my friends Lisa Oliver and Marc Goldford for their encouragement during this project. I'd also like to mention Keith Turner for our many discussions of the proper design, format, and implementation of Java and UML. And thanks also go to my agent, Neil Salkind, for helping with this and other publishing projects.

Finally, I'd also like to acknowledge the great group of people I work with at Reinsurance Group of America Inc. (RGA). I've worked with many people at RGA, but in particular I'd like to mention my colleagues working on CybeRe, namely, Don Apperson, Kam Chan, Terri Dickman, Michael Gaffney, Lisa Hackman, Daniel Horn, Cindy Hsu, Gordon Huffman, Bryan Hunt, Larry Jin, Michel Lefebvre, Melinda Long, Scott Middelkamp, Yuko Miyao, Rick Nolle, Mark Phelps, Renu Singh, Lloyd Sloan, Sharon Stuckmeyer, Matt Tackes, John Tucker, Bala Vanukuru, Jonathan Westland, Thomas Wigington, and anyone else I forgot.

Tell Us What You Think!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As an executive editor for Sams Publishing, I welcome your comments. You can email or write me directly to let me know what you did or didn't like about this book as well as what we can do to make our books better.

Please note that I cannot help you with technical problems related to the *topic* of this book. We do have a User Services group, however, where I will forward specific technical questions related to the book.

When you write, please be sure to include this book's title and author as well as your name, email address, and phone number. I will carefully review your comments and share them with the author and editors who worked on the book.

Email:	feedback@samspublishing.com
Mail:	Michael Stephens Executive Editor Sams Publishing 201 West 103rd Street Indianapolis, IN 46290 USA

For more information about this book or another Sams title, visit our Web site at <http://www.samspublishing.com/>. Type the ISBN (excluding hyphens) or the title of a book in the Search field to find the page you're looking for.

Introduction

JavaServer Pages Standard Tag Library (JSTL) is one of the most significant additions to JSP since the introduction of JSP itself. JSP has traditionally been programmed using scriptlet code—Java source code that is inserted along with the HTML that normally makes up a Web page. JSTL seeks to use tags—similar to those that make up HTML—to replace JSP scriptlet programming. The purpose of this book is to teach you to use JSTL.

First, we must determine exactly what JSTL is. JSTL is a tag library and, as such, is programmed using tags. These tags are similar to the tags you use to program HTML—which is one of the key advantages of JSTL. This similarity is intended is to help alleviate the sharp contrast between the HTML intermingled with JSP scriptlet code. Because JSTL is tag-based, it flows much cleaner with the HTML code.

The Benefits of JSTL

There are several advantages to this tight integration of display and programming tags. First, JSTL is designed so that the Web programmer should be able to learn JSTL quickly and efficiently. JSTL is not just easier for humans to read and understand than JSP scriptlet programming; it is also easier for Web page layout programs to understand. Future Web layout programs will be able to recognize the JSTL tags and insert HTML formatting without disruption of the JSTL tags. Further, Web authoring tools will be able to interact with JSTL tags and perform basic programming.

One perfect example of how JSTL interaction with Web authoring tools can be of particular benefit is in the area of international programming. Traditionally, a multilingual Web application has stored individual text strings in a central location—a database or a data file. The problem is that the method used to display these international text strings is not standardized from one application to the next. Most Web applications use some sort of JSP scriptlet programming to retrieve and then display multilingual text. Unfortunately, Web authoring tools are completely oblivious to these different methods of international programming. JSTL provides us with a concise way to implement a multilingual application. Now it's possible to design Web authoring tools that are aware of the multilingual JSP tags and allows the native language of the page to be displayed just as easily as the foreign languages that the page will be translated into. This enables programmers designing multilingual Web sites to visually edit their Web pages in a variety of languages.

JSTL and JSP

JSTL is designed to work with JSP version 1.2 and the new JSP version 2.0. JSP 2.0 implements some very significant advantages over previous versions of JSP. One of the most important additions to JSP version 2.0 is the expression language, called EL, which is the way that Sun intends future expressions to be represented in JSP programming. JSTL supports this new expression language. For added flexibility, JSTL also supports the expression language typically used in JSP scriptlet programming in versions prior to JSP version 2.0. The EL expression language supports many advanced features.

The EL expression language allows you to access scoped variables. These variables are the primary means by which JSTL stores both temporary and sessionwide data. PCL expressions can be passed to most of the tags provided by the JSTL tag library.

Who This Book Is For

This book assumes that you have some experience as a JSP programmer. An in-depth knowledge of JSP is not required, but some familiarity with HTML forms, hyperlinks, URLs, and server-side programming is assumed. JSTL includes a set of SQL tags that allow the programmer to access a relational database. This book does not attempt to teach SQL, but rather shows you how to use JSTL to integrate with SQL.

Structure and Organization of the Book

This book is designed to be used both as a reference and as a tutorial for the JSTL tag library. The first five chapters discuss the core features of JSTL. If you're not familiar with JSTL, we recommend that you read [Chapters 1 through 5](#) before reading the rest of the book. [Chapters 1 through 3](#) explain core programming techniques that you will need for JSTL programming. [Chapter 4](#) takes an in-depth look at the expression language supported by JSTL. [Chapter 5](#) describes loops and iteration, the core feature of any programming language.

You can read the remaining chapters in any order; they describe individual tags that add specific features to JSTL. [Chapter 6](#) describes how to format data. [Chapter 7](#) examines how to access data with SQL. XML processing is covered in [Chapter 8](#). [Chapter 9](#) discusses Internet programming. [Chapter 10](#) completes the coverage of JSTL and describes how to create an international program.

The final two chapters discuss topics beyond JSTL. [Chapter 11](#) shows how you can create your own tag libraries compatible with JSTL. [Chapter 12](#) demonstrates how to package and deploy your tag-based Web applications.

The Forum Application

One significant feature of this book is the forum application. This application is a Web-based bulletin board that allows users to register and then post messages to any of several forms. Throughout this book, the forum application is extended as you learn new features of JSTL.

We introduce the forum application in [Chapter 7](#), where we implement it as a simple JSP Web application that uses the SQL tags. In the first version, the forum application is an English-only program. [Chapter 10](#) extends this by making the application support additional languages.

The forum application now contains quite a few JSTL tags. JSTL is not meant to be the only technology you use to create a complex Web application; you should use your own custom tag libraries, which you create, to encapsulate your Web application's functionality. [Chapter 11](#) shows you how to create a tag library designed to coexist with JSTL. This allows the forum application tag library to share data with JSTL tags.

At this point, you have a complete Web application that reflects the way in which JSTL should be used in conjunction with your own custom tag libraries. [Chapter 12](#) then completes this process by showing you how to deploy this Web application.

Source Code and Updates

For updates to this book, and to download the source code and examples presented here, visit <http://www.samspublishing.com/>. From the home page, type this book's ISBN (0672324504) into the search window, and click Search to access information about the book and a direct link to the source code.

Chapter 1. Understanding JSP Custom Tags

JavaServer Pages Standard Tag Library (JSTL) is a standard tag library that can be used in JavaServer Pages (JSP). Sun introduced this exciting new technology to allow tag-based programming to exist more naturally with the already tag-based HTML. Because of this, Sun anticipates that JSTL will be an easier programming environment than the traditional scriptlet-based JSP programming. To understand JSTL, you must first understand what a tag library is.

Sun introduced JSTL as part of its Java Community Process (<http://www.jcp.org/>). Most Java Community Process technologies begin as a Java Specification Request (JSR). JSTL began as JSR-52 and JSR-152. JSTL has since been released as JSTL 1.0.

Tag libraries give the Java programmer the ability to write Java objects that will be executed as tags within the JSP code. Custom tags are interpreted by the Web server (or other servlet container) before they are transmitted to the Web browser. Tag libraries can output data of their own, as well as control the flow of HTML and scriptlet code in the JSP page. This allows content to be generated dynamically. Rather than always presenting the same content to the user, a JSP page dynamically creates pages to provide a more interactive experience for the user.

After tag libraries were introduced, many vendors began creating custom tag libraries that could be used with JSP. Many common uses for custom tags were discovered, such as data validation, internationalization, and program flow control. Each vendor implemented these "standard" functions differently.

JSTL is one of several programming methods for creating Web applications using Java. In addition to JSTL, the programmer can use custom tag libraries, scriptlet code, and regular HTML files. These methods are not mutually exclusive, and they can easily be used in conjunction with each other. This chapter presents an overview of these Java Web application technologies and how they fit together.

To help you better understand JSTL, let's step back and review each of the major technologies available to the JSP programmer.

The Role of Tags in JSP

JSP files that use scriptlet code are a mix of HTML tags and Java source code. One of the reasons JSTL and other custom tag libraries were introduced was to provide more consistency in JSP files. HTML is tag-based, whereas the scriptlet code appears as regular Java code. Tag-based programming allows the JSP page to remain completely tag-based. Further, these custom tag libraries have the ability to hide much of the Java

code behind tags that can be inserted directly into the HTML. The result of using JSTL in JSP code is cleaner files that are much easier to modify and extend.

To illustrate the difference between the various programming technologies, let's look at the implementation of a simple example—the process of counting from 1 to 10—in each of these technologies. We begin with an HTML implementation.

Using HTML

HTML is static and cannot change. The look of a page is predefined before the user ever sees it. Listing 1.1 shows how you would count to 10 using just a regular HTML page. Here, you can see that counting to 10 is nothing more than formatting the numbers 1 through 10.

Listing 1.1 HTML Counts to 10

```
<html>
  <head>
    <title>Count to 10 Example (using HTML)</title>
  </head>
  <body>1
  <br />
  2
  <br />
  3
  <br />
  4
  <br />
  5
  <br />
  6
  <br />
  7
  <br />
  8
  <br />
  9
  <br />
  10
  <br />
</body>
</html>
```

Using regular HTML offers a couple of advantages. First, regular HTML is easy to generate. Many GUI tools are available that can be used to generate HTML code. Second, HTML pages can be displayed quickly. The main drawback to HTML pages is that they are static. HTML pages cannot change their appearance as the user works with them. For example, you could not build a shopping cart in HTML only, because you have no way to change the appearance to reflect new items being added.

Using JSP Scriptlets

Now, let's see how the HTML page can be made more dynamic by using JSP scriptlet code. Listing 1.2 shows a page that uses scriptlet code intermixed with HTML. This program also counts to 10.

Listing 1.2 A Scriptlet Counts to 10

```
<html>
  <head>
    <title>Count to 10 Example(using JSP Scriptlet)</title>
  </head>
  <body>
    <%
      for(int i=1;i<=10;i++)
      {
        %><%=i%>
        <br />
        <%
      }
      %>
    </body>
  </html>
```

The scriptlet code has several advantages over HTML. Because scriptlet code is actual Java code, the output from the scriptlet page can be generated as needed. This allows you to create dynamic Web pages that change as the user works with the page. For example, a shopping cart application might use scriptlet code to display the contents of the shopping cart. The primary disadvantage to scriptlet code is that it is not tag-based and does not flow well with the HTML.

Programming in Tags

As you can see, the scriptlet example combines HTML tags with Java source code. Tag programming attempts to combine these two approaches by constructing the scriptlet portion of the file with programmatic tags.

Java is not the only language to try to accomplish Web programming using only tags. Macromedia's ColdFusion allows interactive Web applications to be programmed with tags. Listing 1.3 shows how you would use ColdFusion to count to 10. As you can see, the loop is accomplished by using a special tag, called cfoutput, to perform the actual loop.

NOTE

JSTL source code more closely resembles ColdFusion code than the scriptlet code that most JSP programmers are used to.

Listing 1.3 ColdFusion Counts to 10

```
<html>
  <head>
    <title>Count to 10 Example (using ColdFusion)</title>
  </head>
  <body>
    <cfloop from="1" to="10" index="i">
      <cfoutput>#i#
      <br />
      </cfoutput>
    </cfloop>
  </body>
</html>
```

JSTL-based code resembles ColdFusion code; in both, tags are used for every programming construct, from loops to output. Listing 1.4 shows a JSTL-based page that will count from 1 to 10. As you can see, it resembles the ColdFusion code much more closely than the scriptlet code.

Listing 1.4 JSTL Counts to 10

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
  <head>
    <title>Count to 10 Example (using JSTL)</title>
  </head>
  <body>
    <c:forEach var="i" begin="1" end="10" step="1">
      <c:out value="${i}" />
      <br />
    </c:forEach>
  </body>
</html>
```

Although JSTL is based on Java, JSTL source code does not resemble Java code. You don't necessarily have to decide between scriptlets and JSTL; a single JSP page can intermix both scriptlets and JSTL code. In Chapter 11, "Creating Your Own Tag Libraries," you learn how to create your own tags and use them with both JSTL and scriptlet code.

Another important feature of JSTL is its handling of expressions. JSTL includes an expression language known simply as *Expression Language*, or EL. In Listing 1.4, EL is being used to display the value of the variable i each time through the loop, as seen in the following line:

```
<c:out value="${i}" />
```

The EL expression language is always designated by using the "\${}" notation. The EL expression that is contained between the curly braces will be evaluated. We cover EL in

depth in [Chapter 4](#), "Using the Expression Language." For now, it is enough to know that this line simply prints out the value contained in the variable named `i`.

Because JSTL tags are processed by the Web server, if the user wanted to view the source of [Listing 1.2](#), [1.3](#), or [1.4](#), the output would look very similar to [Listing 1.1](#). This is because the tags are processed at the server, and only the resulting HTML is transmitted. The exact output produced by [Listing 1.4](#) is as follows:

```
<html>
  <head>
    <title>Count to 10 Example (using JSTL)</title>
  </head>
  <body>

    1
    <br />

    2
    <br />

    3
    <br />

    4
    <br />

    5
    <br />

    6
    <br />

    7
    <br />

    8
    <br />

    9
    <br />

    10
    <br />

  </body>
</html>
```

This output contains blank lines between each of the numbers—a side effect of putting the `
` tag on a different line than the `<c:out>` tag. Because you do not see the actual HTML output as you work on your JSP pages, it is easy to place extra space into the HTML output that you would probably not have included if you had "hand-crafted" the HTML file. Generally, this does not affect the file size and is ignored. It is more

important to have easy-to-read, and therefore easy-to-maintain, JSP pages than it is to make sure that you squeeze everything onto just a few lines.

In summary, Sun introduced JSTL to simplify the structure of JSP pages. JSTL improves the appearance of JSP by allowing tag-based code to be used with the HTML. A tag-based programming language fits better with the tag-based HTML.

Using JSTL has its disadvantages, too. The tags provided in JSTL do not allow you to do everything that scriptlet-based JSTL is capable of. Also, some programmers who are more familiar with a procedural programming language may find tags cumbersome when compared to the Java-based code that they are used to.

Another drawback to using JSTL is that additional overhead is introduced. All custom tag libraries, JSTL included, cause the Web server (or other servlet container) to generate additional code to interface to the tag library. In extreme cases when using a considerable amount of tag logic, it is even possible to exceed the maximum Java class size. This extra overhead can slow down your Web application.

Overall, the benefits of JSTL outweigh the drawbacks. Although JSTL is not capable of doing everything that scriptlet-based JSP is capable of, such complex code is often better isolated in an object, away from the presentation-oriented code that should reside in a JSP page.

Introducing JSTL

We now examine the actual structure of JSTL and how to use it. We begin by examining the Apache Taglibs project, which includes the reference implementation of JSTL. Then, we examine the components of JSTL itself.

The Apache Taglibs Project

The implementation of JSTL that we use in this book is a part of the Apache Software Foundation's Jakarta Taglibs project. The Taglibs project includes many tag libraries other than JSTL. All of these tag libraries can be used in JSP pages. You can find more information about the Taglibs project at <http://jakarta.apache.org/taglibs/>. You can use the tag libraries provided by the Taglibs project to add functionality to your JSP pages. However, keep in mind that some of the tag libraries available in the Taglibs project add features that are not currently supported by JSTL.

The Structure of JSTL

JSTL itself is made up of four smaller tag libraries. These tag libraries allow you to choose which components of JSTL you will use in your program. [Table 1.1](#) summarizes the four components of JSTL.

Table 1.1. The Four Components of JSTL

Component	URI	Prefix
Core	http://java.sun.com/jstl/core	c
XML Processing	http://java.sun.com/jstl/xml	x
I18N Formatting	http://java.sun.com/jstl/fmt	fmt
Relational DB Access (SQL)	http://java.sun.com/jstl/sql	sql

Because JSTL includes two expression languages, there are actually eight tag libraries available. To support both the new EL expression language as well as the older scriptlet-based expression language (RT), Sun introduced the concept of "twin tag libraries." To use the scriptlet-based expression language, you must place the suffix-rt on each of the previously mentioned tag libraries. We cover this topic in greater detail in [Chapter 4](#).

The Core Tag Library

The Core tag library contains most of the important tags that you would use for any sort of application. These include the flow control statements, such as if and else. Looping and iteration are also provided by the Core tag library.

We discuss the Core tag library through much of this book. [Chapters 2, 3, 5, and 6](#) specifically cover this library. To use the Core tag library from a JSP page, you must include the following line near the top of your JSP page:

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

The XML Tag Library

The XML tag library provides tags that you can use to access XML documents. Using this tag library ensures that your JSP pages can easily integrate with XML.

[Chapter 8](#), "Accessing Data with XML Tags," specifically addresses the XML tag library. To use the XML tag library from a JSP page, you have to include the following line near the top of your JSP page:

```
<%@ taglib uri="http://java.sun.com/jstl/xml" prefix="x" %>
```

The I18N Tag Library

The I18N (a common abbreviation for internationalization) tag library provides many tags that help with the proper formatting of data from international applications. The I18N tag library allows resource bundles to be used in place of language-specific strings. A *resource bundle* is a collection of language-specific strings that the application can swap with the resource bundle for another language.

We examine the I18N tag library in [Chapter 10](#), "Understanding JSTL Internationalization." To use the I18N tag library from a JSP page, you must include this line near the top of your JSP page:

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

The Relational DBTags (SQL) Library

Generally, it is not considered good programming practice to directly access a database from JSP pages. This is because it is often better to keep something that is as system-dependent as database access out of the presentation layer of the Web application. Therefore, database access is almost always relegated to backend components, such as Enterprise JavaBeans (EJB).

Yet you might still have reason to access SQL directly from a JSP page: for example, for the purpose of debugging or prototyping. JSTL provides a rich set of tags that you can use to access and manipulate any SQL-compatible database.

If you want information about the database tag library, see [Chapter 7](#), "Accessing Data with SQL Tags." To use this library from a JSP page, be sure to include the following line near the top of your JSP page:

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

Installing JSTL

Before you can use the JSTL components, you must have a JSP 1.2-compliant Web server installed, such as Tomcat 4. Installation on other JSP Web servers should be similar. The source code we present in this book should work with any JSP Web server.

If you are not familiar with the installation of Apache Tomcat, you can refer to [Appendix B](#), "Installing Tomcat and JSTL." In this appendix, we tell you how to acquire and install both Tomcat and JSTL.

Adding JSTL to Your Web Server

You must download and install JSTL into your Web server if you want to use the JSTL tags. The process for installing JSTL is similar to the process for installing any other tag library.

You can download JSTL from the Apache Taglibs Project site at <http://jakarta.apache.org/taglibs/>. From the Apache site, choose to download the Standard Tag Library. Again, the complete process for installing JSTL is covered in [Appendix B](#).

Using JSTL in Your JSP Pages

After you have properly installed JSTL into your Web server, you can use it in any of your JSP pages. However, the JSTL libraries do not just become automatically available to each of your JSP pages. Any JSP pages that need to make use of one of the JSTL tag libraries must include an appropriate taglib directive at the top of the JSP page. As we've pointed out, the taglib directive used to access the Core taglib is as follows:

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

With this directive included near the top, this JSP page is now ready to use JSTL taglibs. It is also necessary to make the correct additions to the web.xml file and ensure that JSTL is properly installed, as we explain in [Appendix B](#).

The following is a typical JSTL tag:

```
<c:forEach var="i" begin="1" end="10" step="1">
```

The first part of this tag identifies the tag to be used. In this case, we are using the tag `<c:forEach>`. The part of the tag immediately before the colon specifies the tag library prefix to be used. This allows multiple tag libraries to have tags named the same thing. Here, the c means that we are using the JSTL Core tag library. It would be perfectly valid to use a tag library with the prefix of d that also contained a `<forEach>` tag. Using the prefixes prevents tag library collisions.

Following the tag name are the parameters for this particular tag type. These parameters are specified as XML attributes, as shown in the code snippet. In our example, the forEach tag includes the parameters var, begin, end, and step. The number and exact names of these parameters will vary, depending on the individual tag you are using.

Like every XML tag, JSTL tags must connect with an ending tag. The forEach tag that we just examined must end with the following statement:

```
</c:forEach>
```

XML tags allow you specify additional information, or other tags, between the beginning and ending tags. This practice is used extensively in JSTL. Any tags or data that occur in-between the `<c:forEach>` and its ending tag will be executed repeatedly as part of the forEach tag. The overall structure of such tags is as follows:

```
<c:forEach var="i" begin="1" end="10" step="1">
    ...
</c:forEach>
```

Not every JSTL tag requires a beginning and an ending tag. Another common JSTL tag is the output, or `<c:out>` tag. This tag allows you to output data to the browser. The

`<c:out>` tag is commonly used to print out variables or data retrieved from a database. The following output tag prints out the contents of the variable `i+1`:

```
<c:out value="${i+1}" />
```

As with XML, we are required to put the trailing `/` on the tag because this tag does not include an ending tag. All the formatting rules that apply to XML also apply to JSTL tags.

The simple output tag can be combined with the `forEach` tag. In the following code snippet, these two tags are used together:

```
<c:forEach var="i" begin="1" end="10" step="1">
    <c:out value="${i}" />
</c:forEach>
```

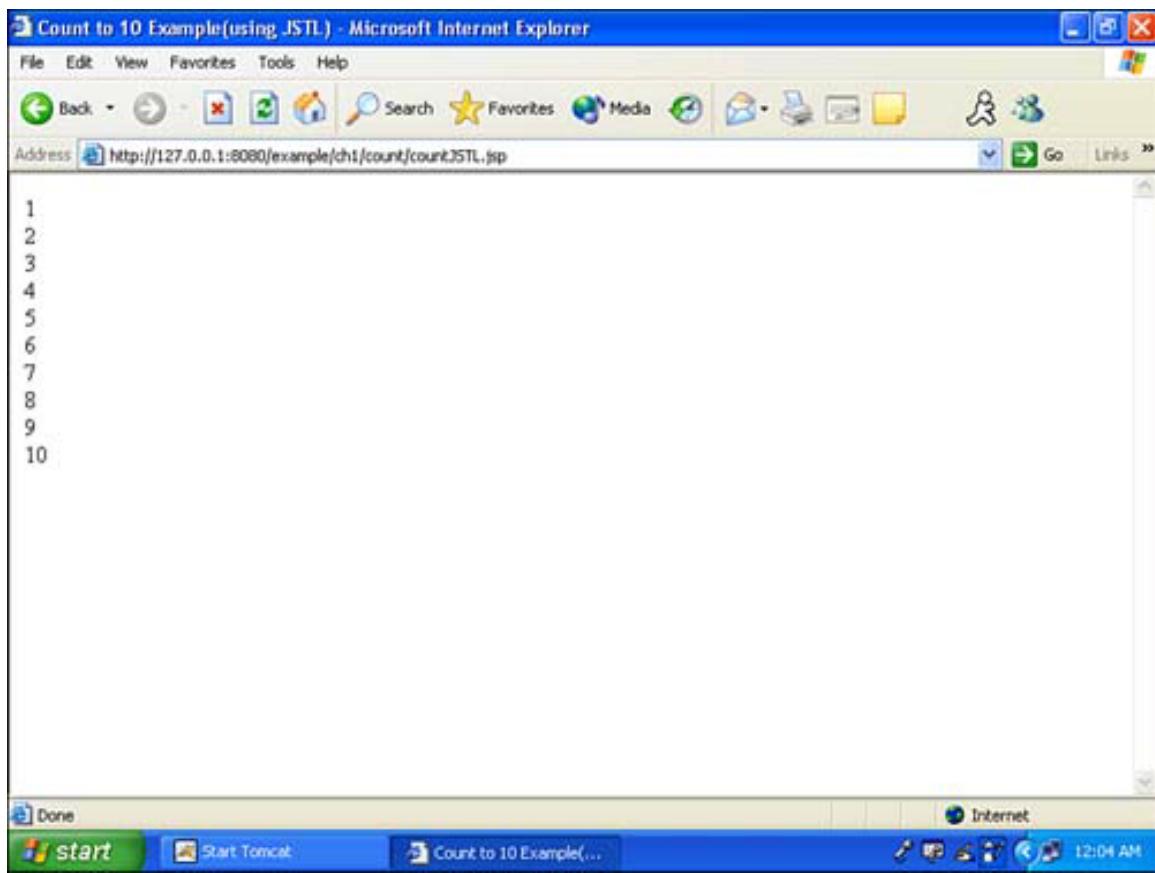
Testing the Counting Example

Now that you have seen the basics of constructing a JSP page, you are ready to execute the counting example. This example, as well as the others in this text, can be freely downloaded from the Sams Web site (<http://www.samspublishing.com/>). You can find these examples in a directory called `example`. We suggest you simply copy the `example` directory in its entirety to your Web root directory. The `example` directory contains the file that corresponds to [Listing 1.4](#), named `countJSTL.jsp`. If you start Tomcat, you should be able to access the file. The URL used to display this example is

`http://127.0.0.1:8080/example/ch1/count/countJSTL.jsp`.

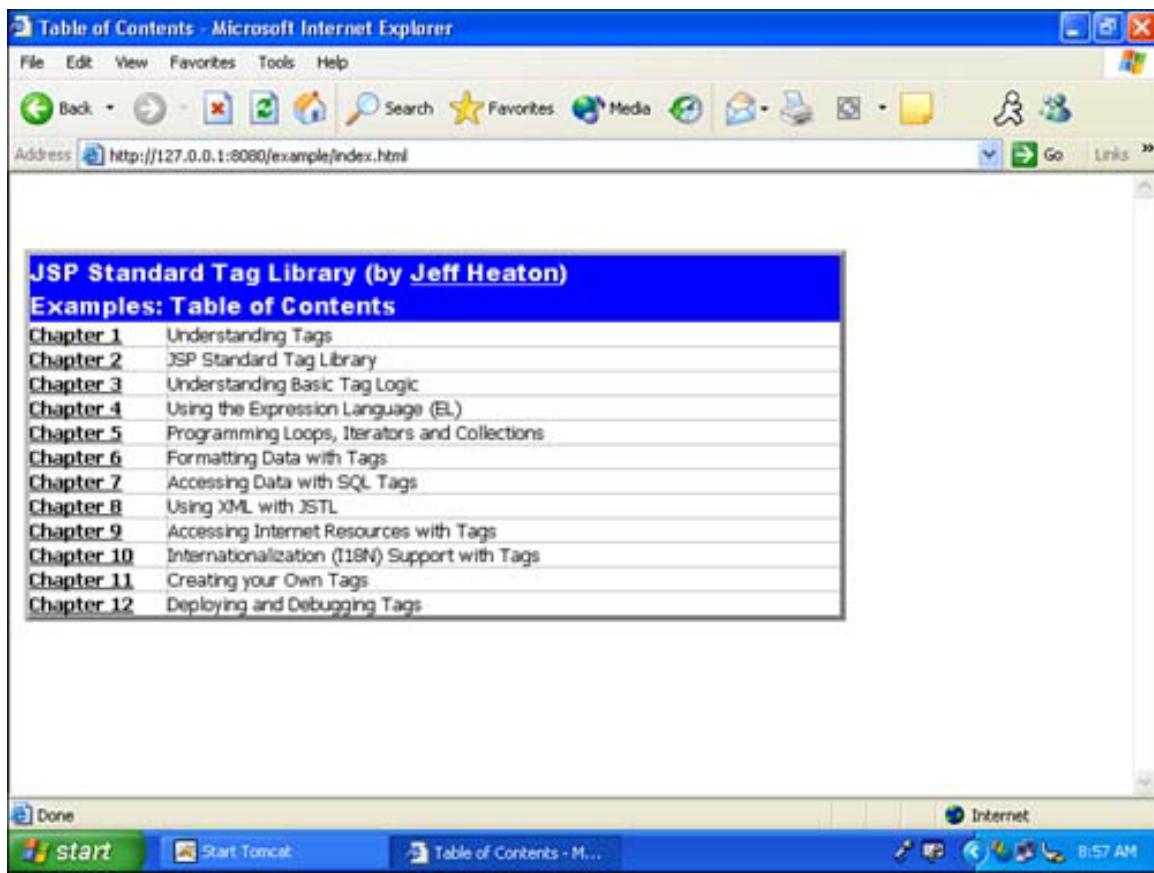
[Appendix B](#) describes how to install as well as use Tomcat in conjunction with Apache. If all goes well, you should see the page shown in [Figure 1.1](#).

Figure 1.1. The counting example.



It is not necessary to enter the exact URL of every example in the book. We provide an index at both the chapter and book level. To view an index of all chapters and links to their examples, navigate to the URL <http://127.0.0.1:8080/example>. You will see the index shown in [Figure 1.2](#).

Figure 1.2. The example index.



Troubleshooting the Counting Example

You have to copy numerous files to various locations to ensure that the counting example will work properly. If you run into problems, reread the previous section to ensure that you followed the steps correctly. If you did, you are ready to run the counting example and most other examples from this book. [Appendix B](#) contains more specific information about troubleshooting problems related to installing JSTL.

Summary

This chapter provided a high-level overview of JSTL and tags in general. The next chapter begins the process of showing you how to create programs in JSTL. We cover such basic concepts as variables and scope.

Chapter 2. Programming the JSTL Standard Tag Library

Every programming language has many fundamental constructs. One of the basic constructs inherent in every programming language is variables. Programming languages must provide variables to store and process data. This chapter begins by showing you how variables are stored and processed in JSTL.

Programs must also interact with the user. When a JSP page first loads all user input to that page, it sends the input as variables. In this chapter, we show you how to access and process this data. This will allow you to create HTML forms that are capable of sending their data to JSTL-based JSP pages.

Form processing is only one aspect of Web application development. Web application programmers must also manage state and provide continuity between their pages. Most Web programming environments accomplish this through the use of some sort of scoped variables. *Scope* refers to both how long a variable remains active and what parts of a program have access to the variable. Scope allows you to define certain variables that are active only for the current page, others that remain active while the user is logged in, and still other variables that are accessible to all users.

Scope is an inherent concept to JSTL variables; in fact, variables in JSTL are called *scoped variables*. As you learn in the next section, there are four levels to JSTL scope.

We conclude this chapter by showing you how to accomplish common Web application tasks with JSTL. All these actions are facilitated by variables.

Understanding Scoped Variables

Most JSTL tags operate on scoped variables. You create JSTL variables by using the set command. A typical set command is shown here:

```
<c:set var="variableName" value="the value"/>
```

This command assigns the value the value to the scoped variable, which is named variableName . If you are familiar with JSP scriptlet programming, you may be interested to know that the scoped variables in JSTL are stored by the same mechanism as JSP scriptlet scoped variables. Just as in JSP scriptlet programming, these variables are stored using the javax.servlet.jsp.PageContext class. Because of this, JSTL scoped variables are accessible to JSP using the PageContext class.

Because JSTL variables are called scoped variables, the concept of scope must enter the picture at some point. The programming term *scope* refers to the scope for which a

variable is valid. In Java, the scope of an automatic, or local, variable is only within the function that initializes the variable. In the following code snippet, the variable `i`'s scope is only within the `mymethod()` method:

```
void mymethod()
{
    int i;
    for(i=0;i<10;i++)
    {
        System.out.println("Count:"+i);
    }
}
```

However, JSTL does not use methods. The scope of a JSTL variable cannot be determined from its context in the same way a Java variable can. Therefore, you must specify the scope of every variable you create. If you fail to specify the scope for a variable, that variable's scope will default to page-level scope. The following tag will create a variable named `variableName` for session-level scope:

```
<c:set var="variableName" value="the value" scope="session"/>
```

JSTL and scriptlet code support four different levels of scope—the same four levels of scope that JSP provides. These four levels are page scope, request scope, session scope, and application scope. [Table 2.1](#) summarizes these scopes.

<i>Table 2.1. Variable Scopes</i>		
Scope	Description	Overhead
Page	For the current page	Very low
Request	For the current request	Low
Session	For the current session	High
Application	Accessible everywhere	Low

Let's examine each level of scope and the differences among them. We begin with page scope.

Page Scope

Page scope is the closest that JSTL comes to automatic, or local, variables. If a variable is created with page scope, its value is valid only inside the current page. If the current page includes (using the JSP include directive) other pages, the page-scoped variable will not be available in these included pages. Further, the page-scoped variable loses its value after the page is displayed to the user. If this user requests that the same page be loaded again, the page-scoped variables are reset.

Page-scoped variables are specific to each load, or display, of the page. They should be used whenever possible because they present the least amount of overhead to the Web

server. Programmers typically use page-scoped variables to hold temporary counters and other variables while a page processes.

To create a page-scoped variable, use the following syntax:

```
<c:set var="scopeVarPage" value="Page Value" scope="page" />
```

The scope attribute specified on the previous line is optional because page is the default scope. Let's look at a sample program that demonstrates the differences in page, request, session, and application scopes. Listing 2.1 shows the main page for the scope example.

Listing 2.1 The Main Scoped Sample Index (index.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<c:set var="scopeVarPage" value="Page Value" scope="page" />
<c:set var="scopeVarRequest" value="Request Value"
scope="request" />
<c:set var="scopeVarSession" value="Session Value"
scope="session" />
<c:set var="scopeVarApplication" value="Application Value"
scope="application" />
<html>
  <head>
    <title>Scope Example</title>
  </head>
  <body>
    <h3>Main File: index.jsp</h3>
    <table border="1">
      <tr>
        <th>Scoped Variable</th>
        <th>Current Value</th>
      </tr>
      <tr>
        <td>
          <b>Page Scope</b>
          (scopeVarPage)</td>
        <td>&#160;
          <c:out value="${scopeVarPage}" />
        </td>
      </tr>
      <tr>
        <td>
          <b>Request Scope</b>
          (scopeVarRequest)</td>
        <td>&#160;
          <c:out value="${scopeVarRequest}" />
        </td>
      </tr>
      <tr>
        <td>
          <b>Session Scope</b>
          (scopeVarSession)</td>
        <td>&#160;
          <c:out value="${scopeVarSession}" />
        </td>
      </tr>
    </table>
  </body>
</html>
```

```

        </td>
    </tr>
    <tr>
        <td>
            <b>Application Scope</b>
            (applicationVarPage)</td>
            <td>&#160;
                <c:out value="${scopeVarApplication}" />
            </td>
        </tr>
    </table>
    <br />
    <br />
    <jsp:include page="included.jsp" />
    <br />
    <br />
    <a href="linked.jsp">[Click Here to View: linked.jsp]</a>
</body>
</html>

```

When the user first views [Listing 2.1](#), four scoped variables are created. The variables `scopeVarPage`, `scopeVarRequest`, `scopeVarSession`, and `scopeVarApplication` are each defined with their respective scope. Next, the values of each variable are displayed when the user first loads this page.

This sample program works by displaying these four variables at different stages in the program. The scope of a variable can be determined by seeing how long it holds its values. The main `index.jsp` page shown in [Listing 2.1](#) uses the following JSP include directive:

```
<jsp:include page="included.jsp" />
```

Often, you will find that you want to reuse JSP code. To allow code to be shared, JSP includes a directive that allows a page to insert the code of another JSP page. To the user, this is seamless, and the included page's HTML is inserted directly into the page that contains the include statement. Of course, an included file can contain JSP.

[Listing 2.2](#) shows an example called `included.jsp`. As you can see, [Listing 2.2](#) displays the values. This allows you to see which of the scoped variables hold their value during the include. Observe that the page-scoped variable loses its value, as seen in [Figure 2.1](#).

Listing 2.2 The Included Scoped Example (included.jsp)

```

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

<table border="1">
    <tr>
        <th>Scoped Variable</th>
        <th>Current Value</th>
    
```

```

</tr>
<tr>
  <td>
    <b>Page Scope</b>
    (scopeVarPage) </td>

    <td>&#160;
      <c:out value="${scopeVarPage}" />
    </td>
  </tr>

<tr>
  <td>
    <b>Request Scope</b>
    (scopeVarRequest) </td>

    <td>&#160;
      <c:out value="${scopeVarRequest}" />
    </td>
  </tr>

<tr>
  <td>
    <b>Session Scope</b>
    (scopeVarSession) </td>

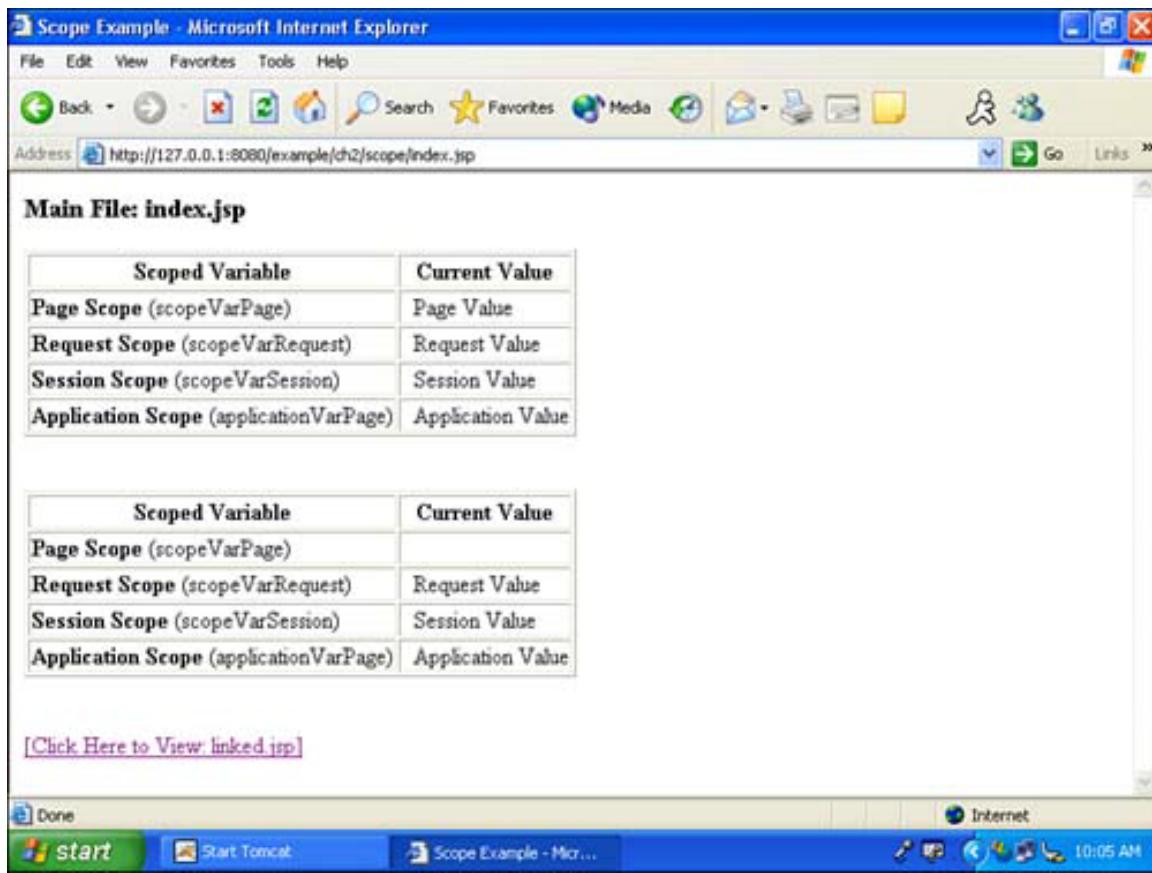
    <td>&#160;
      <c:out value="${scopeVarSession}" />
    </td>
  </tr>

<tr>
  <td>
    <b>Application Scope</b>
    (applicationVarPage) </td>

    <td>&#160;
      <c:out value="${scopeVarApplication}" />
    </td>
  </tr>
</table>

```

Figure 2.1. Running the scope example.



The `index.jsp` file shown in [Listing 2.1](#) also gives the user the opportunity to link to a second page using a hyperlink. The following code provides a link to a second JSP page called `linked.jsp`, which is shown in [Listing 2.3](#):

```
<a href="linked.jsp">[Click Here to View: linked.jsp]</a>
```

Moving to the linked page allows you to see which variables hold their values. Observe that the page-scoped variable loses its value on a linked page as well, as seen in [Figure 2.2](#).

Listing 2.3 The Linked Scoped Example (linked.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
    <head>
        <title>Scope Example</title>
    </head>

    <body>
<h3>Linked File: linked.jsp</h3>

<table border="1">
    <tr>
        <th>Scoped Variable</th>
```

```

<th>Current Value</th>
</tr>

<tr>
<td>
<b>Page Scope</b>

(scopeVarPage)</td>

<td>&#160;
<c:out value="${scopeVarPage}" />
</td>
</tr>

<tr>
<td>
<b>Request Scope</b>

(scopeVarRequest)</td>

<td>&#160;
<c:out value="${scopeVarRequest}" />
</td>
</tr>

<tr>
<td>
<b>Session Scope</b>

(scopeVarSession)</td>

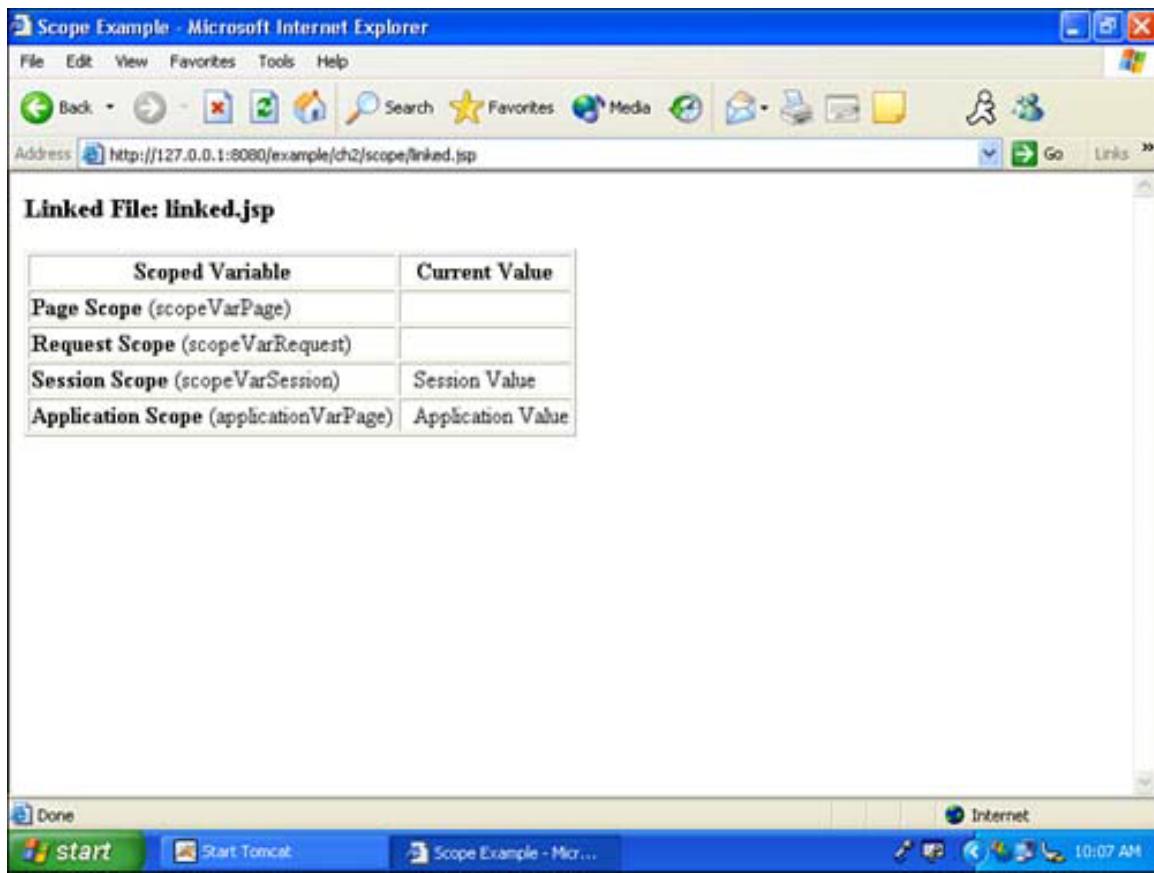
<td>&#160;
<c:out value="${scopeVarSession}" />
</td>
</tr>

<tr>
<td>
<b>Application Scope</b>
(applicationVarPage)</td>

<td>&#160;
<c:out value="${scopeVarApplication}" />
</td>
</tr>
</table>
</body>
</html>

```

Figure 2.2. Examining the linked page.



Figures 2.1 and 2.2 show the output from these two pages. The browser in Figure 2.1 is viewing the `index.jsp` page, which includes both Listings 2.1 and 2.2. Figure 2.2 is showing only `linked.jsp` (Listing 2.3). As you can see, page scope is somewhat limited. On the other hand, the request scope, which we examine next, holds its values over a somewhat larger scope.

Request Scope

A variable that is created with request scope will remain valid for the entire request. A request begins when a Web browser requests a single page, and ends when the Web server has transmitted that page. If you examine Figure 2.1, you will note that the request-scoped variable maintains its value in the included page, yet does not hold its value to the linked page. This is because the linked page is covered by a second request. When users click the hyperlink that takes them to the linked page, that click is a second request.

Request-scoped variables do not introduce much additional overhead beyond the page-scoped variables. If you need to store variables that will be accessible to pages imported with the `include` statement, you must use request-scoped variables. The following line of code creates a request-scoped variable:

```
<c:set var="scopeVarRequest" value="Page Value" scope="request" />
```

Session Scope

Session-scoped variables remain valid for the duration of the current Web session. These variables are tied to individual sessions. As you can see in [Figures 2.1](#) and [2.2](#), the session-scoped variables keep their values both in included files and linked files. To fully understand session-scoped variables, you need to know exactly what a session is.

In the following lines of code, we created a cookie named JSESSIONID. This cookie will be transmitted back to the Web server with every request made by this client. The cookie allows the Web server to tie a user's browser to that user's session.

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=ISO-8859-1
Connection: close
Date: Sun, 14 Apr 2002 19:01:40 GMT
Server: Apache Tomcat/4.0.3 (HTTP/1.1 Connector)
Set-Cookie: JSESSIONID=960A80E2D931F2D671843E6D6EC8A834; Path=/
```

A *session* is an area of memory that stores values for each currently logged-in Web browser. Session-scoped variables are stored in this area. The following line of code creates a session-scoped variable:

```
<c:set var="scopeVarSession" value="The Value" scope="session" />
```

A session ends when one of two things happens. First, a JSP page can specifically terminate a session by calling the HttpSession class's `invalidate()` method. Second, a JSP page can terminate a session using a timeout, which is the method by which most sessions are ultimately terminated. If the session is not used for a predefined amount of time, the session will be discarded. By default, Tomcat uses a 30-minute timeout, but you can change this value in the web.xml file. Use the following lines to set the timeout value:

```
<!-- ===== Default Session Configuration ===== -->
<!-- You can set the default session timeout (in minutes) for -->
<!-- all newly created sessions by modifying the value below. -->

<session-config>
    <session-timeout>30</session-timeout>
</session-config>
```

Because the Web server must allocate memory for each current Web session, session-scoped variables require considerable overhead. Over-dependence on session-scoped variables can considerably increase the processing needs of your Web application and can lead to degraded performance.

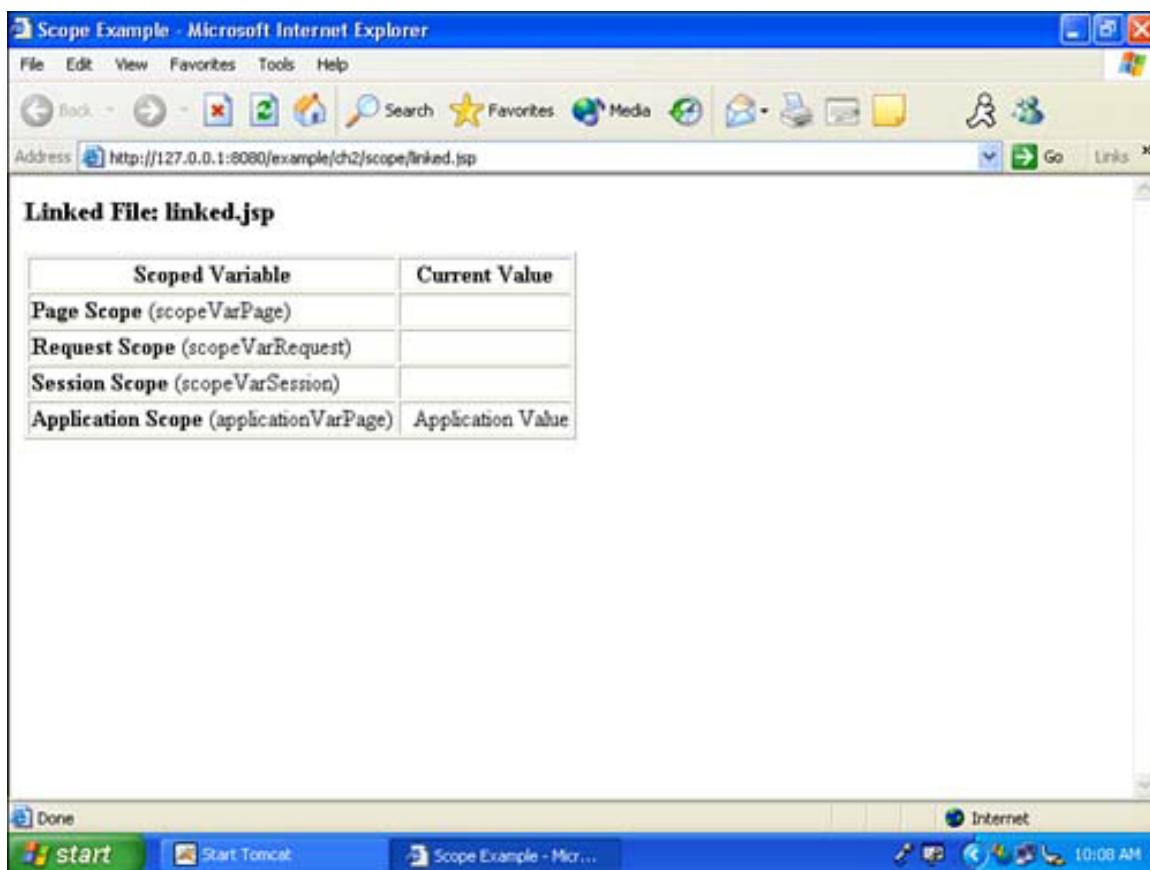
Application Scope

Application-scoped variables are as close as JSTL comes to global variables. When you create an application-scoped variable, it can be accessed from any page and any browser

session. This makes application-scoped variables useful for information that must be shared at a global level by JSP pages. If you look at Figures 2.1 and 2.2, you will notice that application-scoped variables are viewable from all three pages. Notice in Figure 2.1 that session-scoped variables are also accessible in all three locations. So, now we must consider what separates application-scoped variables from session-scoped variables.

If you open a new browser and go directly to the linked page, you will see the difference between session and application variables. By going directly to the linked file, <http://127.0.0.1:8080/example/ch2/scope/linked.jsp>, you bypass the variables being set near the beginning of the index.jsp file. Figure 2.3 shows the results of this direct navigation from a second browser.

Figure 2.3. The application variable from another session.



Notice that the only variable that is set in Figure 2.3 is the application-scoped variable. This is because your new Web browser created a new session. This new session is separate from the previous session; as a result, no session-scoped variables are kept. However, the application-scoped variable remains. This variable will keep its value for every Web session on the Web server. However, if the Web server is restarted, the application-scoped variable will lose its value.

Application-scoped variables consume little overhead because they need to be stored only once and are global to the entire Web application. To create an application-scoped variable, use the following line of code:

```
<c:set var="scopeVarApplication" value="The Value" scope="application">
```

Accessing Application Data

Web applications must interact with the user, and to interact with the user, you must be able to access JSTL application data. Application data falls into two categories in JSTL. JSP makes two implicit variables available for your use.

The first category is page context data. Page context data lets you know general information about the request, such as the request method or how long the session has been running.

The second category, a collection named *params*, is made available to read the content posted from forms. Responding to parameter data is often a primary means of interaction between the Web application and the user.

Accessing Page Context Data

JSP includes a variable named `pageContext` that allows you to access certain contextual information about the page. To see what sort of information is available through the `pageContext` variable, let's look at the example shown in [Listing 2.4](#). This simple JSP page will list the important items stored in the page context.

Listing 2.4 Accessing Application Data

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

<html>
  <head>
    <title>Page Data Example</title>
  </head>

  <body>
    <h3>&#160;</h3>

    <table border="1" width="539">
      <tr>
        <td colspan="2" width="529" bgcolor="#0000FF">
          <b>
            <font color="#FFFFFF" size="4">HTTP
              Request(pageContext.request.)</font>
          </b>
        </td>
      </tr>
    </table>
  </body>
</html>
```

```
</td>
</tr>

<tr>
<td width="210">Access Method</td>
<td width="313">&#160;
<c:out value="${pageContext.request.method}" />
</td>
</tr>

<tr>
<td width="210">Authentication Type</td>

<td width="313">&#160;
<c:out value="${pageContext.request.authType}" />
</td>
</tr>

<tr>
<td width="210">Context Path</td>

<td width="313">&#160;
<c:out value="${pageContext.request.contextPath}" />
</td>
</tr>

<tr>
<td width="210">Path Information</td>

<td width="313">&#160;
<c:out value="${pageContext.request.pathInfo}" />
</td>
</tr>

<tr>
<td width="210">Path Translated</td>

<td width="313">&#160;
<c:out value="${pageContext.request.pathTranslated}" />
</td>
</tr>

<tr>
<td width="210">Query String</td>

<td width="313">&#160;
<c:out value="${pageContext.request.queryString}" />
</td>
</tr>

<tr>
<td width="210">Request URI</td>

<td width="313">&#160;
<c:out value="${pageContext.request.requestURI}" />
</td>
</tr>
```

```

</table>

<table border="1" width="539">
  <tr>
    <td colspan="2" width="529" bgcolor="#0000FF">
      <b>
        <font color="#FFFFFF" size="4">HTTP
          Session(pageContext.session.)</font>
      </b>
    </td>
  </tr>

  <tr>
    <td width="210">Creation Time</td>

    <td width="313">&#160;
      <c:out value="${pageContext.session.creationTime}" />
    </td>
  </tr>

  <tr>
    <td width="210">Session ID</td>

    <td width="313">&#160;
      <c:out value="${pageContext.session.id}" />
    </td>
  </tr>

  <tr>
    <td width="210">Last Accessed Time</td>
    <td width="313">&#160;
      <c:out value="${pageContext.session.lastAccessedTime}" />
    </td>
  </tr>

  <tr>
    <td width="210">Max Inactive Interval</td>

    <td width="313">&#160;
      <c:out
        value="${pageContext.session.maxInactiveInterval}" />
        seconds</td>
  </tr>

  <tr>
    <td width="210">You have been on-line for</td>

    <td width="313">&#160;
      <c:out
        value="${(pageContext.session.lastAccessedTime-
pageContext.session.creationTime)/1000}" />
        seconds</td>
  </tr>
</table>
</body>
</html>

```

As you can see in Listing 2.4, this program displays in the browser many of the values contained in the page context. You can easily access the page context by using the `<c:out>` tag. Use the following line to display the query string for a page:

```
<c:out value="${pageContext.request.queryString}" />
```

We examine query strings in the section "Processing the Query String," later in this chapter. For now, just be aware that you can access the query string only from within the page context. You can see other page context values in Figure 2.4, which shows the output from Listing 2.4.

Figure 2.4. Accessing page context data.

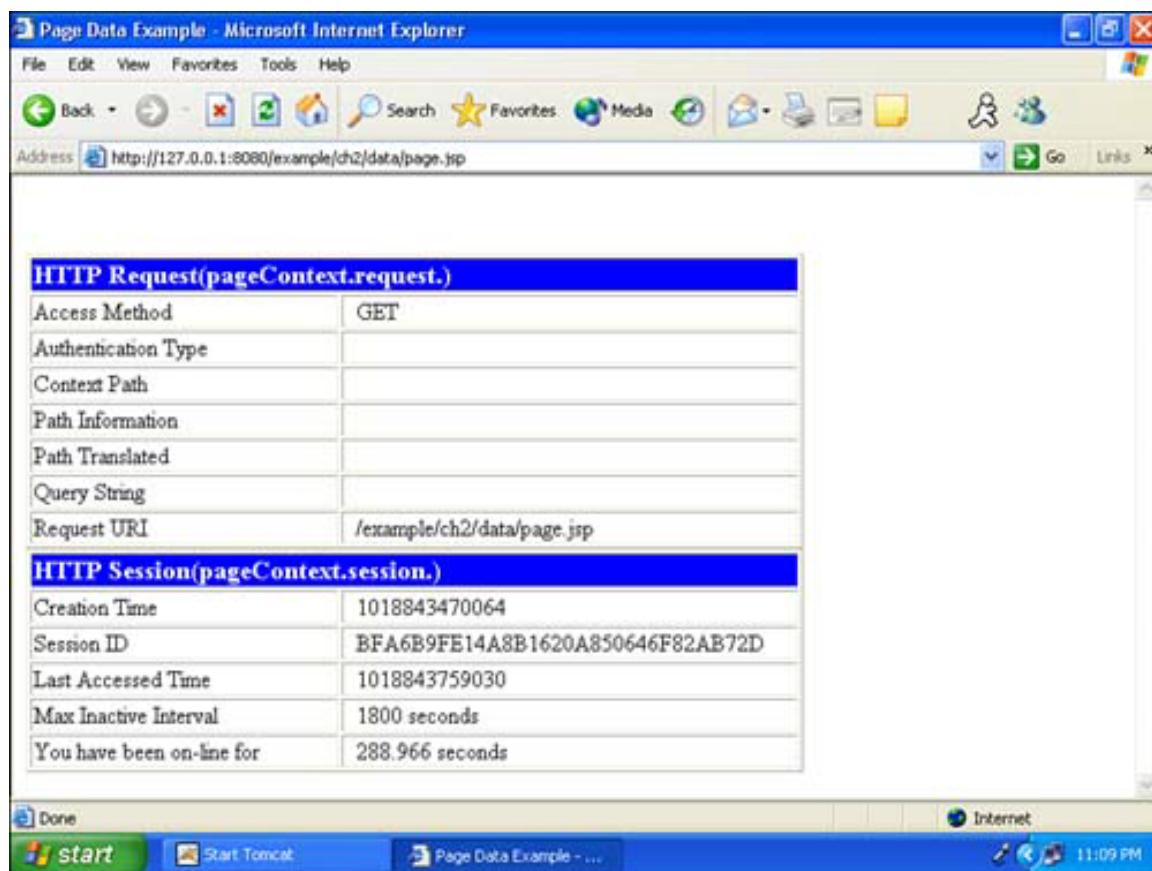


Figure 2.4 shows that the page context data is broken up into the groups *request* and *session*. The request data is accessed using the object `pageContext.request`; the session data is accessed using the `pageContext.session` object.

The following two tables list the most important request and session properties. Table 2.2 lists the request page context properties, and Table 2.3 lists the session page context properties.

Table 2.2. Request Page Context Objects

Property	Purpose
<code>#{pageContext.request.authType}</code>	The authentication type (for example, SSL).
<code>#{pageContext.request.contextPath}</code>	The portion of the request URI that indicates the context of the request. Usually an empty string.
<code>#{pageContext.request.method}</code>	The request method (GET/HEAD/POST).
<code>#{pageContext.request.pathInfo}</code>	Returns extra path information passed to the script (test.jsp/path/).
<code>#{pageContext.request.pathTranslated}</code>	Returns extra path information passed to the script (test.jsp/path/) translated to a real path.
<code>#{pageContext.request.queryString}</code>	The query string from the URL.
<code>#{pageContext.request.requestURI}</code>	The URI portion of the current URL. This is the URL minus the hostname and query string.

Table 2.3. Session Page Context Objects

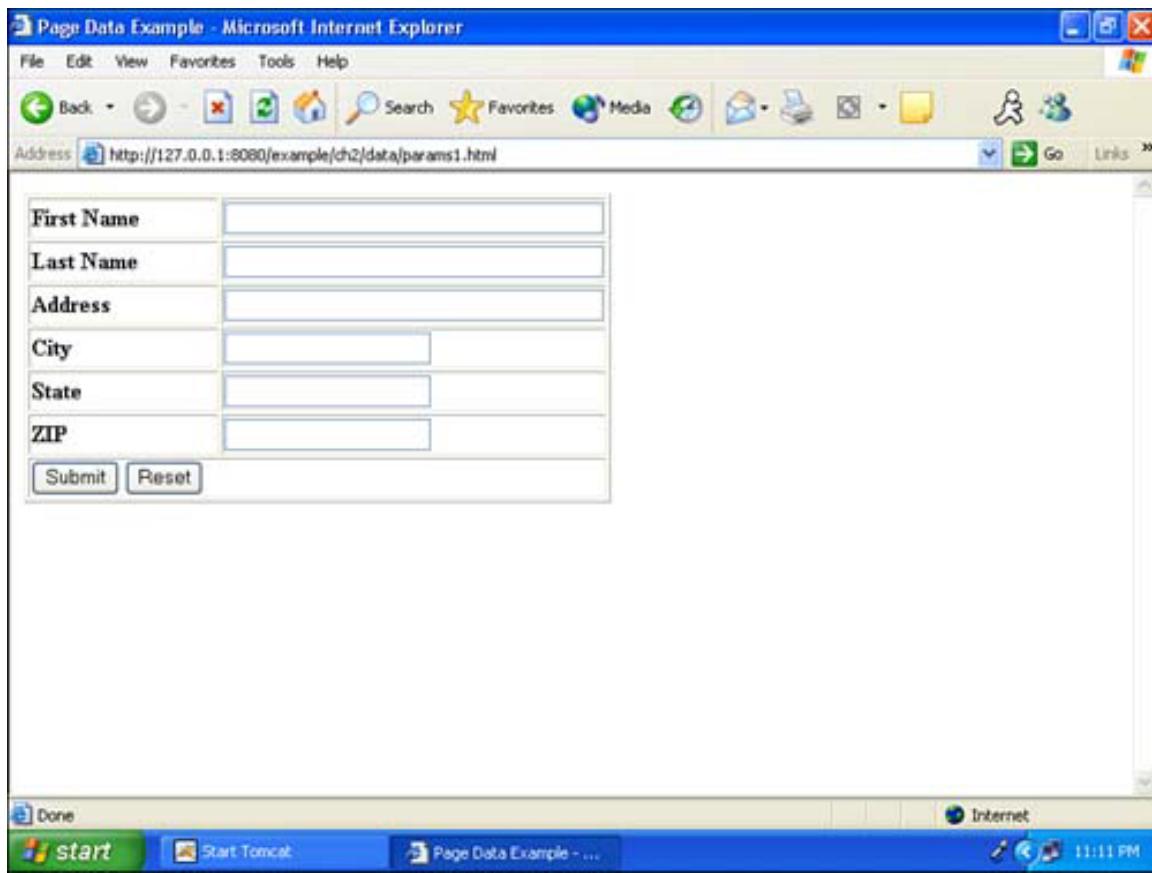
Property	Purpose
<code>#{pageContext.session.creationTime}</code>	The time (in milliseconds since 1/1/1970) that this session was created.
<code>#{pageContext.session.id}</code>	The ID of this session.
<code>#{pageContext.session.lastAccessedTime}</code>	The time (in milliseconds since 1/1/1970) that this session was last accessed.
<code>#{pageContext.session.maxInactiveInterval}</code>	The number of milliseconds before a session is timed out.

Session and request properties give you access to information about the context in which the page was last ran. However, programmers are generally more interested in data that is retrieved from user forms that are posted to your application. In the next section, we show you how to process form data that has been posted to your application.

The `paramValues` Collection

Using the `paramValues` collection, you can access the form data that a user submitted to your page. To understand how to process data from an HTML form, we must first examine the structure of a form. The form that we use for this example appears in [Figure 2.5](#).

Figure 2.5. An HTML form.



The form in [Figure 2.5](#) requests that users enter their name and address. This form is stored in the file params1.html. An HTML file cannot process data by itself; the form on the HTML page simply collects data and posts it to a file that can process data, such as a JSP page. This HTML file is configured to post the data entered into its form to a JSP page named `params2.jsp`. You can see the structure of the HTML form in [Listing 2.5](#).

Listing 2.5 An HTML File That Creates a Form (params1.html)

```
<html>
  <head>
    <title>Page Data Example</title>
  </head>

  <body>
    <table border="1">
      <form method="POST" action="params2.jsp">
        <tr>
          <td width="33%">
            <b>First Name</b>
          </td>

          <td width="73%">
            <input type="text" name="first" size="40" />
          </td>
        </tr>
```

```
<tr>
  <td width="33%">
    <b>Last Name</b>
  </td>

  <td width="73%">
    <input type="text" name="last" size="40" />
  </td>
</tr>

<tr>
  <td width="33%">
    <b>Address</b>
  </td>

  <td width="73%">
    <input type="text" name="address" size="40" />
  </td>
</tr>

<tr>
  <td width="33%">
    <b>City</b>
  </td>

  <td width="73%">
    <input type="text" name="city" size="20" />
  </td>
</tr>

<tr>
  <td width="33%">
    <b>State</b>
  </td>

  <td width="73%">
    <input type="text" name="state" size="20" />
  </td>
</tr>

<tr>
  <td width="33%">
    <b>ZIP</b>
  </td>

  <td width="73%">
    <input type="text" name="zip" size="20" />
  </td>
</tr>

<tr>
  <td colspan="2">
    <input type="submit" value="Submit" name="action" />
    <input type="reset" value="Reset" name="action" />
  </td>
</tr>
```

```
</tr>
</form>
</table>
</body>
</html>
```

As you can see in [Listing 2.6](#), there are several HTML tags that you can use to create a form. You begin a form with this tag:

```
<form method="POST" action="params2.jsp">
```

This form tag contains several important pieces of information. The method attribute specifies how this data is to be sent to the Web server. This is usually either POST or GET. POST is almost always used because compared to GET, it can transfer larger amounts of data.

The second attribute, named action , is used to specify what action the browser should take after the user clicks the Submit button. Here, we can see that the results of this post will be sent to the page `params2.jsp`.

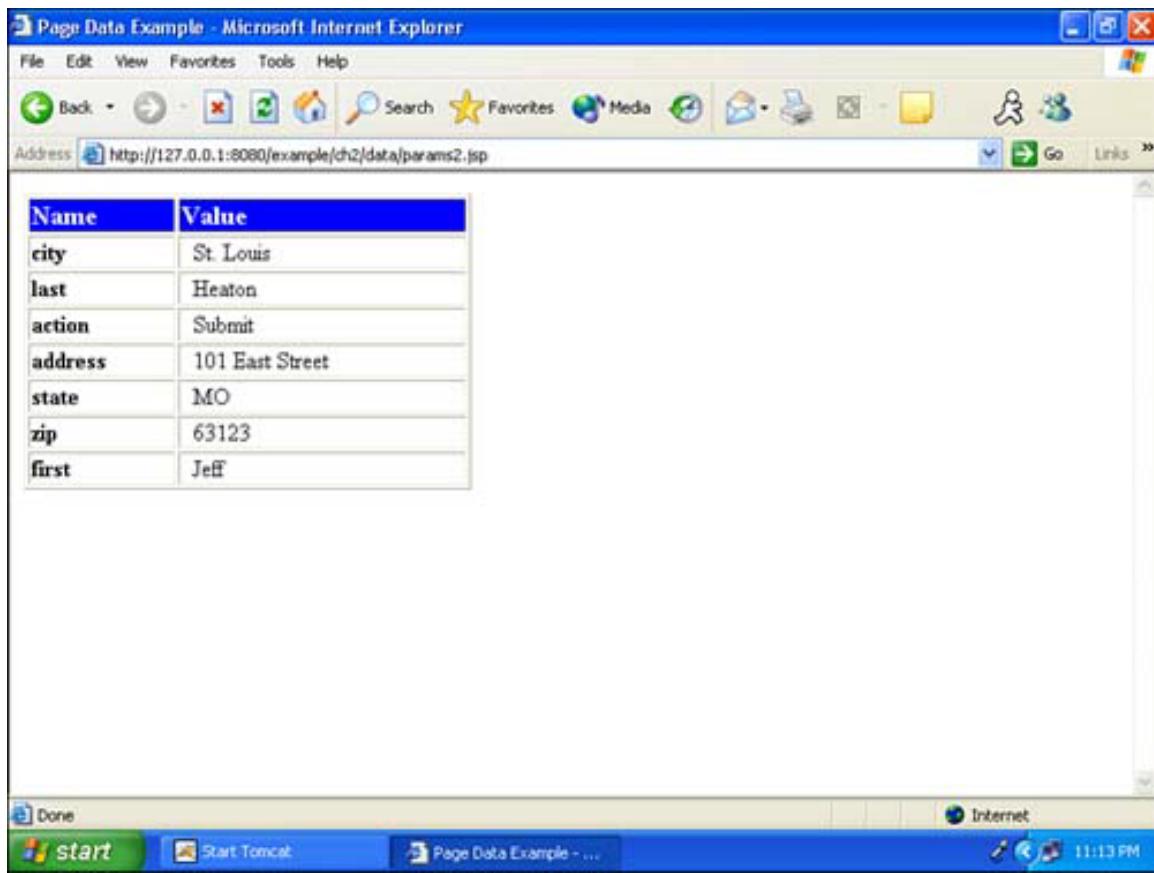
Internally, the form consists of input elements. Each input element will have its entered value posted to the page that was specified in the action attribute of the form element. Consider the following input item:

```
<input type="text" name="state" size="20" />
```

This input item presents itself as a text input item. A text input item provides a blank area into which the user can enter text, as seen in [Figure 2.5](#). When users finish entering data, they click the Submit button, which causes the form to post all its data to the page specified in the action attribute of the form tag.

The action attribute of our form specified that the file `params2.jsp` will handle the form data. When the Submit is complete, `params2.jsp` loads your browser, and you should see something like the display shown in [Figure 2.6](#).

Figure 2.6. Display form data.



As you can see in [Figure 2.6](#), your data is formatted as a table. The file `params2.jsp` was built to display all form input items that it is sent. If you were to add an input item to `params1.html`, a row would automatically be generated in `params2.jsp`. [Listing 2.6](#) shows the `params2.jsp` file.

Listing 2.6 Accessing Form Data (params2.jsp)

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

<html>
  <head>
    <title>Page Data Example</title>
  </head>

  <body>
    <table border="1" width="310">
      <tr>
        <td bgcolor="#0000FF" width="98">
          <b>
            <font color="#FFFFFF" size="4">Name</font>
          </b>
        </td>

        <td bgcolor="#0000FF" width="196">
          <b>
```

```

        <font color="#FFFFFF" size="4">Value</font>
    </b>
</td>
</tr>

<c:forEach var="aItem" items="${paramValues}">
    <tr>
        <td width="98">
            <b>
                <c:out value="${aItem.key}" />
            </b>
        </td>

        <td width="196">&#160;
            <c:forEach var="aValue" items="${aItem.value}">
                <c:out value="${aValue}" />
            </c:forEach>
        </td>
    </tr>
</c:forEach>
</table>
</body>
</html>

```

The `params2.jsp` file works by listing every item that is in the `paramValues` collection. This is done with a `<c:forEach>` tag, which is used to access every element in a collection. We discuss the `<c:forEach>` tag in much greater detail in [Chapter 5](#), "Collections, Loops, and Iterators." The following code sets up the file to read every item in the `paramValues` collection:

```
<c:forEach var="aItem" items="${paramValues}">
```

This code causes the variable `aItem` to contain each of the form properties one by one. If a form contains more than one element with the same name, elements of the same name will form an array. The `params` collection is used to access these element arrays. Consider if the following two form elements were submitted:

```

<input type="text" name="state" size="20" />
<input type="text" name="state" size="20" />

```

Both form elements have the name `state`. Consequently, the form element for `state` would be a collection that contains two items. The code used to loop through an input item's collection (designated by `aItem`) is shown here:

```

<c:forEach var="aValue" items="${aItem.value}">
    <c:out value="${aValue}" />
</c:forEach>

```

Usually, the form elements that you access are individual elements rather than arrays. The usual case is when you seek input items with specific names. See the section "Working with Forms" later in this chapter for more information. Now, let's see how the

paramValues collection can be used for types to retrieve information from the query string as well as forms.

The paramValues collection does not just process forms; it also receives information from the query string. For example, the URI `script.jsp?a=valuea&b=valueb` contains parameters named a and b, contained on the query string.

There are other implicit variables that you may find helpful as well. You can access variables from the page, request, session, and application scopes with the variables `pageScope`, `requestScope`, `sessionScope`, and `applicationScope`.

The Basics of Web Application Programming

Now that you have a good understanding of JSTL variables and application data, you can start to create some simple Web applications. This section provides examples that illustrate common Web programming techniques.

Processing the Query String

The query string has always been a popular conduit for information between Web pages. The query string allows variable information to be embedded directly into a URL. For example, the URL `http://www.jeffheaton.com/search.jsp?n=5` would pass the value 5 through the n variable to the script `search.jsp`. Anything containing a ? that follows the URL is considered part of the query string.

By separating the values with ampersands (&), you can send more than one value through the query string. To pass the value 10 for x and 20 for y, you would use the URL `http://www.jeffheaton.com/search.jsp?x=10&y=20`.

These values can easily be accessed by the JSTL code. You will recall from the section "Accessing Application Data" earlier in this chapter that JSTL provides a variable called paramValues that allows you to access parameter arrays. There is also a variable called param that allows access to individual form and query string items. Note that neither param nor paramValues makes any distinction between form items and query string items. You would use the following tag to display a form or query string item named x :

```
<c:out value="${param.x}" />
```

[Listing 2.7](#) shows a program that uses this technique. This simple example asks users what their favorite color is.

Listing 2.7 Favorite Color (color.jsp)

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

```

<html>
  <head>
    <title>Query String Example</title>
  </head>

  <body>Your favorite color is:
  <b>
    <c:out value="${param.color}" />
  </b>

  <br />
  <br />

  Choose your favorite color:
  <br />

  <br />

  <a href=<c:out value="${pageContext.request.requestURI}" />?color=red">
    red</a>

  <br />

  <a href=<c:out value="${pageContext.request.requestURI}" />?color=blue">
    blue</a>

  <br />

  <a href=<c:out value="${pageContext.request.requestURI}" />?color=green">
    green</a>
  <br />

  <a href=<c:out value="${pageContext.request.requestURI}" />?color=yellow">
    yellow</a>

  <br />

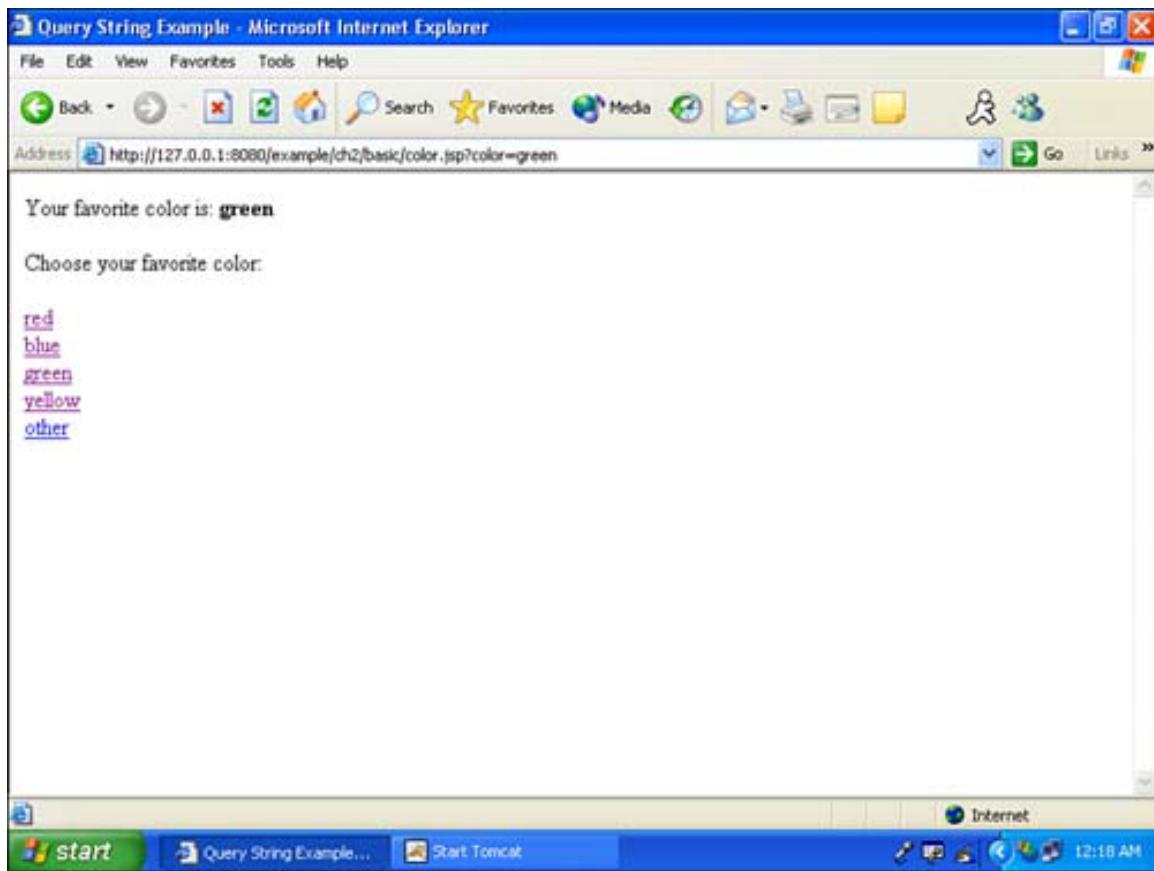
  <a href=<c:out value="${pageContext.request.requestURI}" />?color=other">
    other</a>

  <br />
  </body>
</html>

```

The program in Listing 2.7 displays the current favorite color, which is initially blank. Then, the user is allowed to select from several hyperlinks to specify a new favorite color. You can see the output from this program in Figure 2.7.

Figure 2.7. Pick your favorite color.



This program works by passing the current favorite color on the query string. Notice the hyperlinks that allow users to select their favorite color:

```
<a  
href="yellow</a>
```

These hyperlinks embed a color, such as *yellow*, as part of the URL line. The `requestURI` is also used to ensure that the program links back to the same page. We could have simply used the name `color.jsp` rather than the `requestURI`; however, using the `requestURI` ensures that the code will continue to function, even if the script filename changes. When the user actually clicks the hyperlink, the resulting URL will be something in the form of

`http://127.0.0.1:8080/example/ch2/basic/color.jsp?color=yellow.`

When the user clicks on one of the color selection hyperlinks, the page will be reloaded. However, this time it will have a value for the query string. The program will then display the favorite color using this code:

```
<body>Your favorite color is:  
<b>
```

```
<c:out value="${param.color}" />
</b>
```

Working with Forms

Forms are a common part of any Web application. The param variable allows you to access forms and retrieve individual entries made by the user. In this section, we look at a simple example that asks the user for a user ID and password. The form posts the results to a second JSP page that greets the user. This is a simple example that shows how to retrieve a single value from a form. You will perform this operation many times while creating Web applications. Listing 2.8 shows the form that inputs the user ID and password.

Listing 2.8 Our HTML Login Form (form1.html)

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

  <body>
    <form method="POST" action="form2.jsp">
      <table border="1" cellpadding="0" cellspacing="0"
        style="border-collapse: collapse" bordercolor="#111111"
        width="42%" id="AutoNumber1">
        <tr>
          <td width="100%" colspan="2" bgcolor="#0000FF">
            <p align="center">
              <b>
                <font size="4" color="#FFFFFF">Please Login</font>
              </b>
            </p>
          </td>
        </tr>

        <tr>
          <td width="19%">User Name</td>
          <td width="81%">
            <input type="text" name="uid" size="20" />
          </td>
        </tr>

        <tr>
          <td width="19%">Password</td>
          <td width="81%">
            <input type="password" name="pwd" size="20" />
          </td>
        </tr>

        <tr>
          <td width="100%" colspan="2">
            <p align="center">
              <input type="submit" value="Submit" name="action" />
            </p>
          </td>
        </tr>
      </table>
    </form>
  </body>
</html>
```

```

        <input type="reset" value="Reset" name="B2" />
    </p>
</td>
</tr>
</table>
</form>

<p align="left">
    <i>Note: you may use any ID/Password, security is not
    checked.</i>
</p>
</body>
</html>

```

This is a typical form that posts to another page. The user ID and password information is collected from the user and then posted to `form2.jsp`, shown in [Listing 2.9](#).

Listing 2.9 Processing the Login Form (form2.jsp)

```

<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
    <h3>Welcome back
    <c:out value="${param.uid}" />

    !</h3>
</html>

```

This is a simple JSP page that displays the user's ID, stored in the uid input tag, to the browser.

You have seen how forms are processed using JSTL. Forms are an important way of retrieving information from the user, and you will use them often in Web programming.

Understanding State

State is an important concept for nearly any transactional Web site. If the user is to log in, some form of state must be used so that the system remembers which browser represents the user who just logged in. Without state, the Web server will forget a user as soon as he or she logs in. State is what the Web server remembers from one request to the next. In this example, the state would be which user is actually logged on.

To help you better understand state, we have combined the previously mentioned examples. This program asks the user to log into the Web site. After logging in, the user is prompted to enter his or her favorite color. Unlike in the previous examples, this program remembers the name of the user who is logged in, and greets him or her for each new color picked. The login screen is also somewhat different, as you can see in [Listing 2.10](#).

Listing 2.10 Our Stateful Login Form (session1.jsp)

```

<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<c:if test="${pageContext.request.method=='POST'}">
    <c:set var="uid" value="${param.uid}" scope="session" />

    <jsp:forward page="session2.jsp" />
</c:if>
<html>
    <head>
    </head>

    <body>
        <form method="POST">
            <table border="1" cellpadding="0" cellspacing="0"
                style="border-collapse: collapse" bordercolor="#111111"
                width="42%" id="AutoNumber1">
                <tr>
                    <td width="100%" colspan="2" bgcolor="#0000FF">
                        <p align="center">
                            <b>
                                <font size="4" color="#FFFFFF">Please Login</font>
                            </b>
                        </p>
                    </td>
                </tr>

                <tr>
                    <td width="19%">User Name</td>
                    <td width="81%">
                        <input type="text" name="uid" size="20" />
                    </td>
                </tr>

                <tr>
                    <td width="19%">Password</td>
                    <td width="81%">
                        <input type="password" name="pwd" size="20" />
                    </td>
                </tr>

                <tr>
                    <td width="100%" colspan="2">
                        <p align="center">
                            <input type="submit" value="Submit" name="action" />
                            <input type="reset" value="Reset" name="B2" />
                        </p>
                    </td>
                </tr>
            </table>
        </form>

        <p align="left">
            <i>Note: you may use any ID/Password, security is not
            checked.</i>
        </p>
    </body>
</html>

```

```
</body>
</html>
```

The first obvious difference is the form tag. As you can see, the `<form method="POST">` tag does not specify an action.

If you do not specify an action for a form tag, the form will post to the current page. This may sound like an endless loop, and without JSTL it would be. However, at the top of `session1.jsp` are a few lines of code that intercept the post, as shown here:

```
<c:if test="${pageContext.request.method=='POST'}">
    <c:set var="uid" value="${param.uid}" scope="session" />
    <jsp:forward page="session2.jsp" />
</c:if>
```

Here, an if statement is used to check the request method. (We cover if statements in greater detail in [Chapter 3](#), "Understanding Basic Tag Logic.") If the request method is POST, this page is currently being posted to. If this is the case, the user ID is read and stored as a session variable named uid. The page then is forwarded to `session2.jsp`, causing the `session2.jsp` page to be displayed. The user will never even know that we briefly revisited `session1.jsp`. The `session2.jsp` file is shown in [Listing 2.11](#).

Listing 2.11 Our Favorite Color Example Using State (session2.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
    <head>
        <title>Session Example</title>
    </head>

    <body>
        <h3>Welcome
            <c:out value="${uid}" />
        </h3>

        Your favorite color is:
        <b>
            <c:if test="${param.color!=null}">
                <c:out value="${param.color}" />
            </c:if>
        </b>
        <br />
        <br />
        Choose your favorite color:
        <br />
        <br />
        <a href=<c:out
value="${pageContext.request.requestURI}"/>?color=red">
            red</a>
        <br />
        <a href=<c:out
value="${pageContext.request.requestURI}"/>?color=blue">
```

```

blue</a>
<br />
<a href=<c:out
value="\${pageContext.request.requestURI}"/>?color=green">
  green</a>
<br />
<a href=<c:out
value="\${pageContext.request.requestURI}"/>?color=yellow">
  yellow</a>
<br />
<a href=<c:out
value="\${pageContext.request.requestURI}"/>?color=other">
  other</a>
<br />
</body>
</html>

```

The file `session2.jsp` is identical to `color.jsp`, except that it greets the current user. It knows who the current user is because of the session-scoped variable. The following lines of code display this variable:

```

<h3>Welcome
<c:out value="\${uid}" />
</h3>

```

For this code, it is not necessary to include the `scope="session"` portion of the command. If you do not specify a session, JSTL will examine the request, page, session, and finally application scopes until a match is found.

State is an important part of Web development. In this section, you learned how to manage the state, which is what the Web server remembers from request to request. You will use state often as you design and implement interactive Web applications.

Summary

This chapter introduced you to the basics of scoped variables and how they are used with JSTL. In the next chapter, we show you how to use basic tag logic to perform if statements and other branching operations.

Chapter 3. Understanding Basic Tag Logic

The Core tag library of JSTL provides many rudimentary tags for performing basic flow control. Flow control refers to a program's ability to selectively execute portions of itself. Most programming languages achieve this flow control through if statements and other logical tags. A program's flow of execution can also be affected by errors.

Error handling is an important part of any computer program. JSP provides tags that let you perform basic error handling. By making use of JSTL's built-in exception-handling capabilities, you can create programs that are capable of handling common runtime errors.

In this chapter, we present a sample application that shows how to tie all these concepts together. A simple forms-based chat application will show you how to use scope and basic tag logic to allow two users to chat. Let's begin by examining JSTL's error-handling capabilities.

Exception Processing

Java programs handle errors through the use of exceptions. When a section of code is susceptible to throwing an exception, you can enclose that section of code in a try block. For example, when you're creating a new URL object, it is possible that an invalid URL will be provided to the object. If this happens, the URL object will throw a MalformedURLException. To properly handle this exception, the URL instantiation must be enclosed in a try block, and a catch must be provided that will catch the MalformedURLException. The following code illustrates this:

```
try
{
    URL = new URL("http://www.sams.com");
}
catch(MalformedURLException e)
{
}
```

Exception handling allows your program to react to errors that are not a result of bad program design. For example, exceptions could be trapped when the user enters invalid information, a URL fails to load, or when some other noncritical error occurs. By trapping these exceptions, you design your program to handle these errors as they happen so they don't result in a server error. This ensures that your Web pages present a consistent and robust interface.

JSTL provides the `<c:catch>` tag to handle this case. Before we examine how to catch an exception in JSTL, let's see what causes an exception to be thrown.

Generating Exceptions

Many of the JSTL tags are capable of throwing exceptions, yet JSTL is designed to throw exceptions as little as possible. Many errors that would generate an exception in Java or other programming languages are ignored in JSTL. Consider division by zero. Nearly any programming language ever produced will return some sort of an error if a division is specified with zero in the denominator. Yet in JSTL, this case does not generate an exception. [Listing 3.1](#) demonstrates this.

Listing 3.1 Division by Zero (exception.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %><html>
  <head>
    <title>Throw an Exception</title>
  </head>

  <body>10 divided by 0 is
  <c:out value="${10/0}" />

  <br />
  </body>
</html>
```

As you can see, [Listing 3.1](#) attempts to throw an exception by dividing 10 by 0. Under most circumstances, this would throw an exception. However, JSTL simply ignores this error and continues processing. As you learn more about JSTL error handling, you will see that in most cases, JSTL attempts to continue processing after an error has occurred. If you run the JSP code in [Listing 3.1](#), you get the following output:

```
10 divided by 0 is infinity
```

While JSTL attempts to handle most exceptions as they occur, certain things can go wrong that will still cause JSTL to throw an exception. [Listing 3.2](#) shows an example of an error that will throw an exception.

Listing 3.2 An Uncaught Exception (exception2.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
  <head>
    <title>Throw an Exception</title>
  </head>

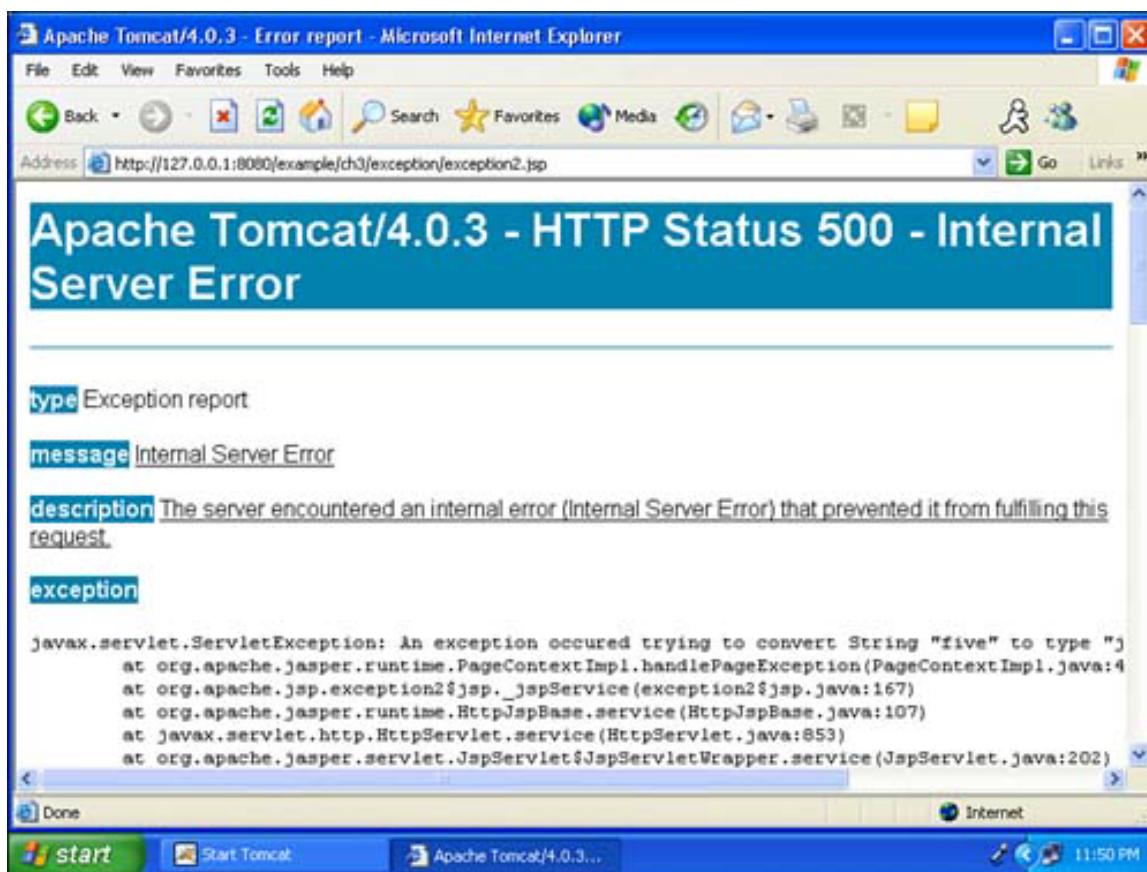
  <body>
  <c:set var="x" value="10" scope="page" />
  <c:set var="y" value="five" scope="page" />

  x divided by y is
  <c:out value="${x/y}" />
```

```
<br />
</body>
</html>
```

[Listing 3.2](#) attempts to throw an exception by causing a type mismatch. The value 10 is stored in the scoped variable *x*. The value five, the literal string, is stored in the scoped variable *y*. The program then attempts to divide *x* by *y*. While it is permissible to divide by zero in JSTL, it is not permissible to divide a string. When executed, [Listing 3.2](#) will throw an exception. Because there is nothing set up to catch this exception, it will be thrown to the Web browser as a server error. [Figure 3.1](#) shows this error message.

Figure 3.1. An uncaught JSTL exception is thrown.



It is important to understand which errors in JSTL will result in an exception and which errors will be ignored. In general, the following errors will cause an exception to be thrown.

- Specification of an invalid scope
- An empty var attribute in a tag
- An invalid type
- An invalid value
- Invalid expressions

JSTL is very forgiving of most other errors. For example, null values, which often cause errors in other languages, are simply treated as empty strings.

Now that you have seen what can cause an exception, it is important to see how to handle exceptions. The next section covers how to catch exceptions in JSTL.

Using the <c:catch> Tag

JSTL provides the `<c:catch>` tag for catching exceptions. Any tags that are within the body of the `<c:catch>` tag will be handled. The `<c:catch>` tag is capable of catching any exception that is a subclass of `java.lang.Throwable`; this includes all Java exceptions. The following shows the structure of a `<c:catch>` tag:

```
<c:catch var="e">
    ... Program code that may thrown an exception ...
</c:catch>
```

One attribute is accepted by the `<c:catch>` tag:

Attribute	Required	Purpose
<code>var</code>	Y	Specifies the scoped variable to receive the exception.

Our previous sample would catch an exception and store it in the `e` variable. If an exception occurs inside the body of the `<c:catch>` tag, JSTL immediately continues executing with the line just after the ending `</c:catch>` tag. The variable specified in the `var` attribute, in this case `e`, will receive a copy of the exception that was thrown. If no exception is thrown, then the exception variable will receive a null value.

When inserting a `<c:catch>` tag in your code, make sure that the code handles both an exception being thrown and an exception not being thrown. Listing 3.3 shows the catch division-by-zero example with a `<c:catch>` tag ready to handle any errors.

Listing 3.3 Our Catch Division-by-Zero example (catch.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
    <head>
        <title>Catch an Exception?</title>
    </head>

    <body>
        <c:catch var="e">

            10 divided by 0 is
            <c:out value="${10/0}" />

            <br />
        </c:catch>
    </body>
</html>
```

```

<c:if test="${e!=null}">The caught exception is:  

<c:out value="${e}" />  
  

<br />
</c:if>  
  

<c:if test="${e==null}">No exception was thrown  

<br />
</c:if>
</body>
</html>

```

As you can see in the following code snippet, the division by zero is now wrapped in a `<c:catch>` tag. If any error occurs, the program will immediately exit the `<c:catch>` tag:

```

<c:catch var="e">
10 divided by 0 is
<c:out value="${10/0}" />
<br />
</c:catch>

```

Because the division by zero does not cause an exception, this program will glide through the catch and continue on with a null value assigned to the variable `e`. The program continues by checking this variable and displaying the error if it was set:

```

<c:if test="${e!=null}">The caught exception is:  

<c:out value="${e}" />  

<br />
</c:if>

```

If an exception does occur, the program stores a value in the variable specified to the `<c:catch>` tag. Listing 3.4 shows an example of a program that will throw an exception. The line that generates the exception is protected by a `<c:catch>` tag. As a result, the exception will be handled internally by the JSP page.

Listing 3.4 Catching an Exception (catch2.jsp)

```

<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
  <head>
    <title>Catch an Exception</title>
  </head>

  <body>
    <c:catch var="e">
      <c:set var="x" value="10" scope="page" />

      <c:set var="y" value="five" scope="page" />

      x divided by y is
      <c:out value="${x/y}" />

    <br />
  </body>
</html>

```

```

</c:catch>

<br />
<c:if test="${e!=null}">The caught exception is:
<c:out value="${e}" />

<br />
</c:if>

<c:if test="${e==null}">No exception was thrown
<br />
</c:if>
</body>
</html>

```

This program performs in exactly the same way as the division-by-zero example. However, in this case, the code following the `<c:catch>` tag actually has something to print. This sample program will display the following:

The caught exception is: An exception occurred trying to convert String "five" to type `java.lang.Double`.

This text string identifies the exception that was thrown, and includes a message that describes why the error occurred.

Using Conditionals

Conditional statements are an important part of any programming language. These statements allow you to selectively execute parts of the program based on certain conditions. JSTL provides several tags that allow this sort of conditional execution.

JSTL's implementation of conditionals is somewhat different from Java's. In Java, your two primary conditional constructs are the if/else statement and the switch/case statement. The `<c:if>` and `<c:choose>` tags provide many of the same features, yet there are some important differences compared to their Java counterparts. We highlight these differences in this section.

In previous chapters, we have seen how the `<c:if>` tag can be used to make basic decisions. We now take a more in-depth look at the `<c:if>` tag and other conditional tags provided by JSTL. Let's begin by examining the `<c:if>` tag.

Using the `<c:if>` Tag

Nearly every programming language contains an if statement, and JSTL is no exception. The JSTL if statement is implemented using the `<c:if>` tag. There are two forms of the JSTL if statement:

```

<!-- Syntax 1: Without body content -->
<c:if test="testCondition"
var="varName" [scope="{page|request|session|application}"]/>

<!-- Syntax 2: With body content -->
<c:if test="testCondition"
[var="varName"] [scope="{page|request|session|application}"]>
body content
</c:if>

```

The attributes that are accepted by the `<c:if>` tag are as follows:

Attribute	Required	Purpose
<code>test</code>	Y	Specifies the test that is to be performed.
<code>scope</code>	N	Specifies the scope; the default is page.
<code>var</code>	N	Specifies the scoped variable that should receive the result of the compare. (Note: This attribute is required if the scope is present or there is no body.)

The second form demonstrates an if statement that encloses a body of code. If the expression given to the if statement evaluates to true, then that body of code is executed. [Listing 3.5](#) shows an example of using the if statement with a body of code.

Listing 3.5 An if Statement with a Body (ifbody.jsp)

```

<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
  <head>
    <title>If with Body</title>
  </head>

  <body>
    <c:if test="${pageContext.request.method=='POST'}">
      <c:if test="${param.guess=='5'}">You guessed my number!
      <br />

      <br />
    </c:if>

    <c:if test="${param.guess!='5'}">You did not guess my number!
    <br />

    <br />
    </c:if>
  </c:if>

  <form method="post">Guess what number I am thinking of?
  <input type="text" name="guess" />

```

```

<input type="submit" value="Try!" />

<br />
</form>
</body>
</html>

```

This program implements a simple number-guessing game. You are presented with a form that allows you to enter a number. When your form posts back to the page, the number you entered is compared against five. If the numbers match, you win; if the numbers do not match, you are given a chance to try again. The format of the if statement is shown here:

```

<c:if test="${param.guess=='5'}">
    You guessed my number!
</c:if>

```

Most expression operators that are available in Java can also be used in a JSTL if statement. Here we are using `==` to request a comparison. Other operators, such as not equal(`!=`), greater than(`>`), less than(`<`), less than or equal(`<=`), and greater than or equal(`>=`), are also supported. With only a few minor exceptions, these operators work exactly the same as they do in Java. The most notable difference is that the equality operator(`==`) can be used to compare String objects. These operators are handled as part of JSTL's expression language. We cover these operators in greater detail in [Chapter 4](#), "Using the Expression Language."

It is also possible to create an if statement that does not contain a body of code to execute. The uses for this "bodyless" statement are somewhat more limited than the general if statement. [Listing 3.6](#) shows a program that makes use of the bodyless if statement.

[Listing 3.6 An if Statement with No Body \(ifnobody.jsp\)](#)

```

<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
    <head>
        <title>If with NO Body</title>
    </head>

    <body>
        <c:if test="${pageContext.request.method=='POST'}">
            <c:if test="${param.guess=='5'}" var="result">

                I tested to see if you picked my number, the result was
                <c:out value="${result}" />
            </c:if>

            <form method="post">Guess what number I am thinking of?
            <input type="text" name="guess" />

            <input type="submit" value="Try!" />
        </c:if>
    </body>
</html>

```

```
<br />
</form>
</body>
</html>
```

In Listing 3.6, the if statement stores the result of its comparison to a variable rather than conditionally executing a block of code. First the if statement is invoked; then the variable is printed.

In the following code snippet, a boolean value is stored in the scoped variable named result. When the value is printed to the browser, either *true* or *false* will be displayed:

```
<c:if test="${param.guess=='5'}" var="result" />
I tested to see if you picked my number, the result was
<c:out value="${result}" />
```

In most programming languages, an else statement is provided to go along with the if statement. JSTL does not provide an else tag. There is a relatively simple way to produce an if/else construct, but it cannot be done with the if statement. To produce an if/else construct, you must use the `<c:choose>` tag. The next section will focus on this tag.

Using the `<c:choose>` Tag

JSTL provides you with a mutual exclusion conditional. The `<c:choose>` tag is called a mutual exclusion because only one block of code within the mutual exclusion will be executed. Therefore, the block of code that is chosen to execute is said to mutually exclude all other blocks of code. There is only one form of the `<c:choose>` tag:

```
<c:choose>
    body content (<when> and <otherwise> subtags)
</c:choose>
```

No attributes are associated with the `<c:choose>` tag. The `<c:choose>` tag is almost always used with the `<c:when>` tag, as we discuss in the next section.

Using the `<c:when>` Tag

In some respects, the mutual exclusion can be compared to the switch/case or if/else tree construct in Java. The `<c:choose>` tag is comparable to the Java switch statement, and the `<c:when>` tag is comparable to the case statements of a switch statement. The general format of the JSTL mutual exclusion is shown here:

```
<c:when test="testCondition">
    body content
</c:when>
```

One attribute is accepted by the `<c:if>` tag:

Attribute	Required	Purpose
test	Y	Specifies the test that is to be performed.

A `<c:choose>` tag is essentially a block of if statements. When a `<c:choose>` block is encountered, the `<c:when>` statements are evaluated in order. Once one `<c:when>` statement is satisfied, it is executed and the `<c:choose>` exits. It does not matter if more than one `<c:when>` statement's test is satisfied. The first `<c:when>` statement that is successful is the only `<c:when>` statement that will be executed.

The biggest difference between the `<c:choose>/<c:when>` tag and the switch/case statements in Java is that the `<c:when>` statements actually evaluate expressions. A switch/case in Java only compares a variable specified in the switch statement against constants in the case statements. This makes the `<c:when>` statements much more advanced than case statements.

[Listing 3.7](#) shows an example of the mutual exclusion in action. This program shows a simple input form that asks you to enter a number. The program then responds by spelling out the name of the first five numbers after one.

Listing 3.7 Mutual Exclusion (choose.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
    <head>
        <title>Using Choose,Otherwise and When</title>
    </head>

    <body>
        <c:if test="${pageContext.request.method=='POST'}">You entered:

        <c:choose>
            <c:when test="${param.enter=='1'}">One
            <br />
            </c:when>

            <c:when test="${param.enter=='2'}">Two
            <br />
            </c:when>

            <c:when test="${param.enter=='3'}">Three
            <br />
            </c:when>

            <c:when test="${param.enter=='4'}">Four
            <br />
            </c:when>

            <c:when test="${param.enter=='5'}">Five
            <br />
            </c:when>
        </c:choose>
    </body>
</html>
```

```

<c:otherwise>
    <c:out value="${param.enter}" />

    <br />
</c:otherwise>
</c:choose>
</c:if>

<form method="post">Enter a number between 1 and 5:
<input type="text" name="enter" />

<input type="submit" value="Accept" />

<br />
</form>
</body>
</html>

```

If you enter a value that is not in the range of one through five, the program prints out your number in numeric form. This default behavior is provided by the `<c:otherwise>` tag, which we will cover in the next section.

Using the `<c:otherwise>` Tag

The `<c:otherwise>` tag provides the functionality of Java's default and else statements. The `<c:otherwise>` tag may only be used as part of a `<c:choose>` block. The following shows the general format of the `<c:otherwise>` tag:

```

<c:otherwise>
    conditional block
</c:otherwise>

```

No attributes are associated with the `<c:otherwise>` tag. If you use only one `<c:when>` tag with a `<c:otherwise>` tag, you have created something that is functionally equivalent to the if/else construct. Listing 3.8 shows how the `<c:otherwise>` tag can be used to provide an "else" statement.

Listing 3.8 An "if" Statement with "else" (ifelse.jsp)

```

<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
    <head>
        <title>Using Choose, Otherwise and When</title>
    </head>

    <body>
        <c:if test="${pageContext.request.method=='POST'}">OK, we'll
        send
        <c:out value="${param.enter}" />

        <c:choose>
            <c:when test="${param.enter=='1'}">pizza.

```

```

<br />
</c:when>

<c:otherwise>pizzas.
<br />
</c:otherwise>
</c:choose>
</c:if>

<form method="post">Enter a number between 1 and 5:
<input type="text" name="enter" />

<input type="submit" value="Accept" />

<br />
</form>
</body>
</html>

```

The code in [Listing 3.8](#) asks in a form how many pizzas you would like to order. The program then prints the message "OK, we'll send xxx pizza(s)." The if/else construct is used to cause the program to print out the word *pizza* if only one pizza is being ordered and to print out the word *pizzas* if more than one is ordered. The following block of code implements the if/else construct:

```

<c:choose>
  <c:when test="${param.enter=='1'}">
    pizza.
  </c:when>
  <c:otherwise>
    pizzas.
  </c:otherwise>
</c:choose>

```

As you can see, a `<c:when>` tag is used to evaluate whether the order is equal to one. If the order is equal to one, then the program prints the word *pizza*. If the order is not equal to one, then the word *pizzas* is printed.

Nesting Conditionals

Like any other programming language, JSTL allows you to nest conditionals. All JSTL tags are XML-style tags. You nest conditionals in JSTL by nesting their XML tags. For example, the following shows two if statements nested:

```

<c:if test="${pageContext.request.method=='POST'}">
  <c:if test="${param.guess=='5'}">
    You guessed my number!
  </c:if>
</c:if>

```

Conditionals can be nested very deep. You can also nest different conditionals within each other. For example, it is completely valid to nest a `<c:if>` tag inside a `<c:when>` tag.

A Chat Application

The last example that we examine in this chapter is a simple chat application. This program shows how more than one browser session can be made to communicate. We use a simple example of a chat application in order to demonstrate some of the techniques shown in this chapter. [Chapter 5](#), "Collections, Loops, and Iterators," will extend this simple chat program into a more complex example.

When you first access the chat page, you will be prompted to enter your login identity. This is the name by which other users will know you. This login can be nearly anything you like. Once you enter your name, you will be taken to the main chat page. (See [Figures 3.2](#) and [3.3](#).)

Figure 3.2. Our first user chatting.

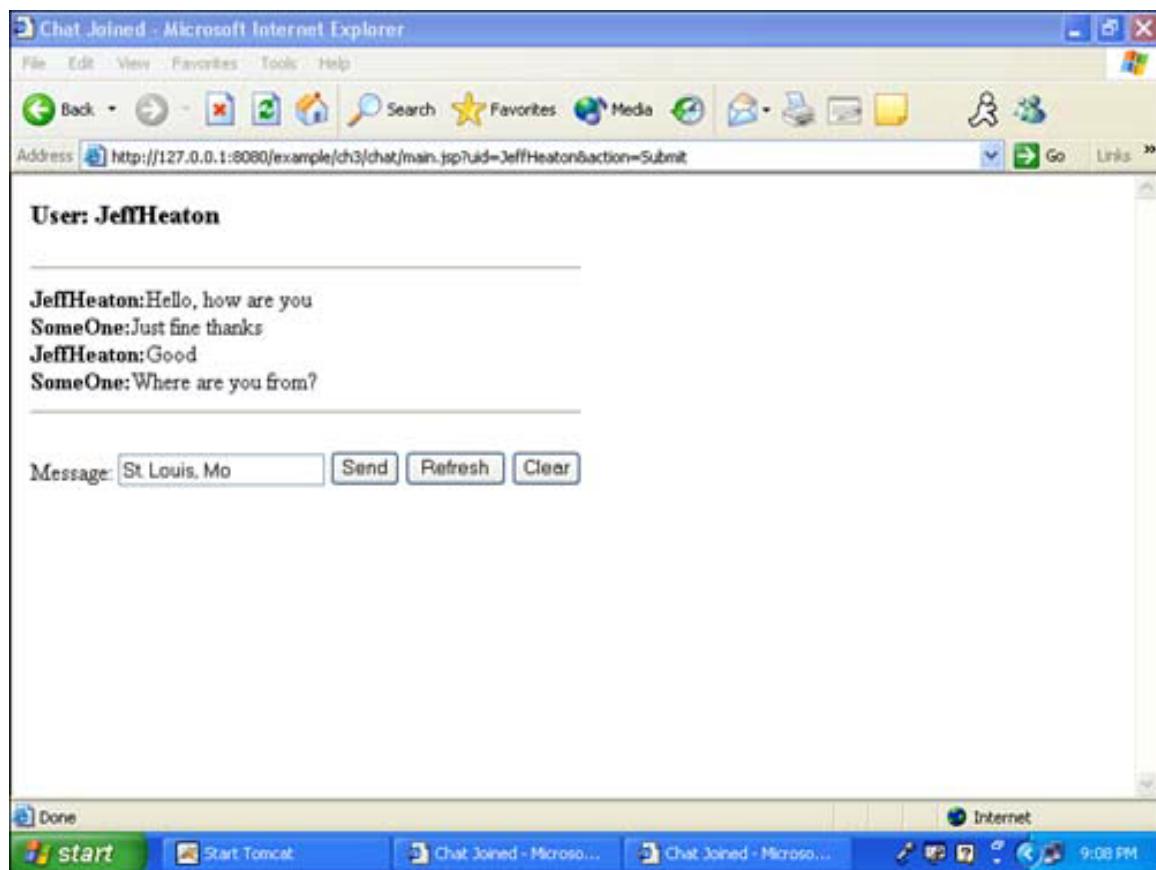
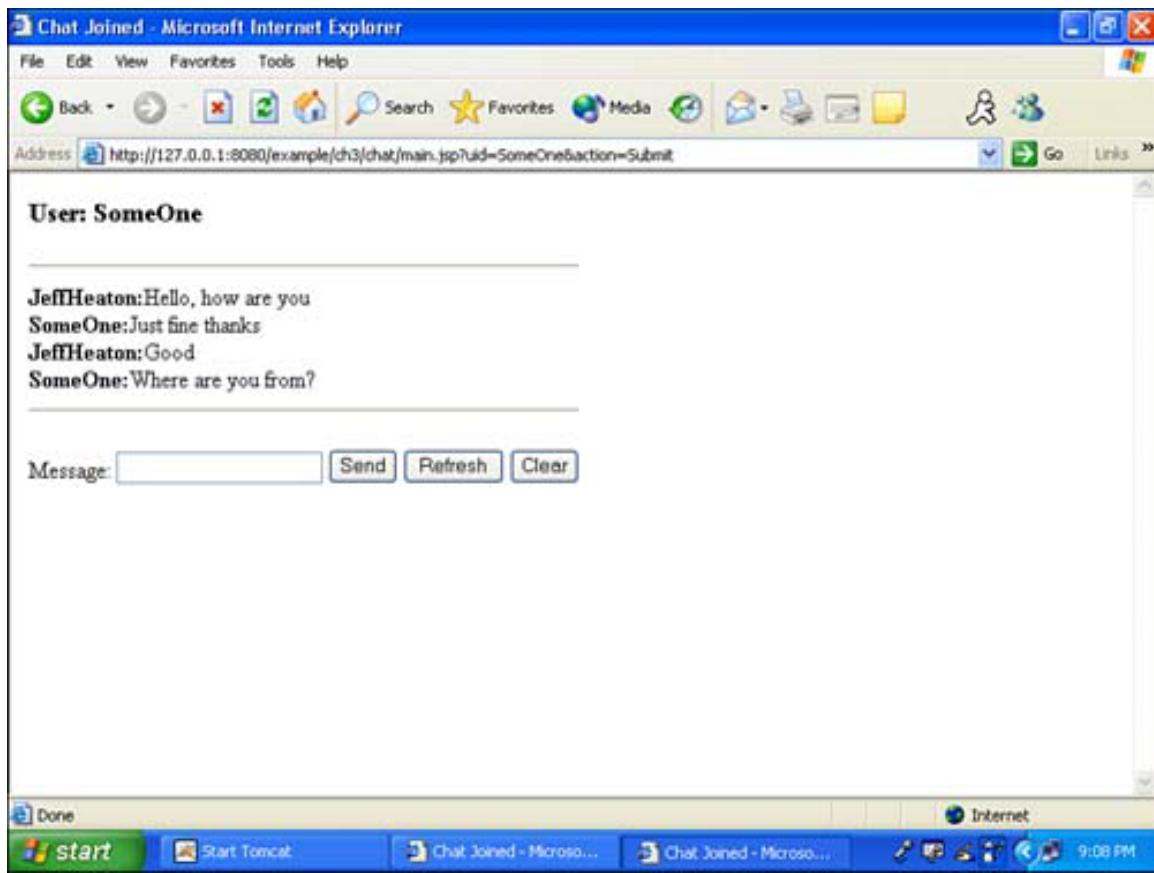


Figure 3.3. Our second user chatting.



The login page for the chat application is a typical HTML form. This file appears in [Listing 3.9](#).

Listing 3.9 Our Chat Login (index.html)

```
<html>
  <head>
    <meta http-equiv="Content-Language" content="en-us" />

    <title>Simple Chat Application</title>
  </head>

  <body>
    <form action="main.jsp">
      <table border="1" cellpadding="0" cellspacing="0"
        style="border-collapse: collapse" bordercolor="#111111"
        width="33%" id="AutoNumber1">
        <tbody>
          <tr>
            <td width="100%" colspan="2" bgcolor="#0000FF">
              <p align="center">
                <b>
                  <font size="4" color="#FFFFFF">Chat Login</font>
                </b>
              </p>
            </td>
```

```

</tr>

<tr>
    <td width="23%">User ID</td>

    <td width="77%">
        <input type="text" name="uid" size="20" />
    </td>
</tr>

<tr>
    <td width="100%" colspan="2">
        <p align="center">
            <input type="submit" value="Submit"
                   name="action" />

            <input type="reset" value="Reset" />
        </p>
    </td>
</tr>
</tbody>
</table>

<p>&#160;</p>
</form>

<p>
    <i>Note: You may use any User ID you wish</i>
</p>
</body>
</html>

```

Most of the action for this program takes place on the main chat screen. Here, you have three options. First, you can send your message. This will make your message visible to the other users, including yourself. Second, you can refresh your view of the screen. As other users enter messages, they will not immediately be visible to you. If you wish to update your screen without sending a message, you must click the Refresh button. Third, a Clear button is provided that gives you the ability to clear the display. Because messages never scroll off the chat, clicking Clear lets you start from a clean slate. Listing 3.10 shows the source code for the main chat screen.

Listing 3.10 Our Main Chat Page (main.jsp)

```

<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
    <head>
        <title>Chat Joined</title>
    </head>

    <body>
        <c:if test="${pageContext.request.method=='POST'}">
            <c:choose>
                <c:when test="${param.send!=null}">
                    <c:set var="chat"

```

```

        value="\${chat}<b>\${param.uid}:</b>\${param.text}<br />" 
        scope="application" />
</c:when>

<c:when test="\${param.clear!=null}">
    <c:set var="chat" value="" scope="application" />
</c:when>
</c:choose>
</c:if>

<table border="0">
    <tbody>
        <tr>
            <td>
                <h3>User:</h3>
                <c:out value="\${param.uid}" />
            </td>
            <hr />
        </td>
    </tr>

    <tr>
        <td>
            <c:out value="\${chat}" escapeXml="false" />
        </td>
    </tr>

    <tr>
        <td>
            <hr />
            <form method="post">Message:
                <input type="text" name="text" size="20" />
                <input type="submit" name="send" value="Send" />
                <input type="submit" name="refresh" value="Refresh" />
                <input type="submit" name="clear" value="Clear" />
                <input type="hidden" name="uid"
                    value="" />
            <br />
            </form>
        </td>
    </tr>
</tbody>
</table>
</body>
</html>
```

This program does not store the username in a session-scoped variable, as previous examples have. Instead, the username is stored as a hidden form field that is posted back to the main chat page every time you click a button.

The contents of the chat are stored in an ever-increasing application-scoped variable. When you click the Send button, the message you just typed is added to this application-scoped variable via the following:

```
<c:set var="chat"  
value="${chat}<b>${param.uid} :</b>${param.text}<br />"  
scope="application" />
```

This code concatenates the new message onto the messages that already exist in the application-scoped variable. Because the chat message is stored at the application level, the only way to clear the chat area is to either restart the Web server or click the Clear button. Clicking Clear executes the following line of code, which clears the chat application-scoped variable:

```
<c:set var="chat" value="" scope="application" />
```

Clicking the Refresh button simply redraws the window without adding any messages. The chat area, when redrawn, is displayed with the following line of code:

```
<c:out value="${chat}" escapeXml="false" />
```

You will notice that a property called escapeXml is set to false. Without this property set to false, JSTL would automatically encode the HTML tags that we are adding to the chat. This would be bad because the program would display these tags rather than processing them. For example, the HTML tag `
` would be encoded to be `
`; this would prevent our formatting effects from displaying.

In summary, the chat application is a simple JSTL application that makes use of some of the most basic tags in the JSTL tag library. The primary purpose of these tags is to control the flow of information through the chat application. This flow of information begins with the form, where users enter their message. The form is then posted back to itself, and `<c:choose>` and `<c:when>` tags are used to see if a conversation already exists. Then a `<c:set>` tag is used to store the conversation to an application-scoped variable so that all sessions can see the conversation. Finally, this conversation is presented to the user along with a form the user can use to add to the conversation. This completes the cycle.

Summary

In earlier chapters, you have seen expressions used in conjunction with the techniques being introduced. The next chapter will focus on expressions and show you some of the advanced features they provide.

Chapter 4. Using the Expression Language

Programming languages use expressions to manipulate application data. Expressions resemble algebraic formulas in that they often specify mathematical transformations involving the program's variables. For example, the expression $a+10$ tells the program to evaluate the expression 10 added to a . This does not change the value of a ; rather, the computer simply computes this sum and then performs some action based on that sum.

JSTL supports expressions through the use of two expression languages. JSTL includes support for scriptlet-based expressions, called `rtpexprvalue` expressions. JSTL also introduces Expression Language (EL), which greatly simplifies expression handling in JSTL. The designers of JSTL assume that most programmers will use EL expressions and use `rtpexprvalues` for more advanced purposes. This chapter will focus primarily on EL expressions, although we cover expressions using `rtpexprvalues` as well. We begin by examining the tags that JSTL makes available for expression handling.

JSTL Expression Tags

In JSTL, an expression is always wrapped between the characters \${ and }. For example, the expression for displaying the variable a with 10 added would be \${ $a+10$ }. This can easily be incorporated into many of the JSTL tags. The following code would use a `<c:out>` tag to display the expression $a+10$:

```
<c:out value="${a+10}" />
```

Eventually, JSP 2.0 will allow EL expressions to be inserted right into HTML code. For example, the following line in a JSP file would display the expression $a+10$:

```
<p>The value of a+10 is ${a+10}</p>
```

While the expression language is a powerful feature of JSP 2.0, one of the design requirements for JSTL was compatibility with JSP 1.2. To accommodate JSP 1.2's lack of expression support, Sun added the `<c:out>` tag to provide a means of displaying a JSTL expression.

The `<c:out>` tag is not the only tag that JSTL provides that can work with expressions. In addition to the `<c:out>` tag, JSTL includes several tags that are designed to deal directly with expressions. In the following sections, we explain how to use each of these tags.

Using the <c:out> Tag

The `<c:out>` tag is used to display a JSTL expression and set the value of a JSTL scoped variable. There are two forms of the `<c:out>` tag:

```
// Syntax 1: Without a body
<c:out value="value" [escapeXml="{true|false}"]
[default="defaultValue"] />
// Syntax 2: With a body
<c:out value="value" [escapeXml="{true|false}"]>
default value
</c:out>
```

If you want to display a small amount of text, you should use the bodyless version of the `<c:out>` tag. This will cause your code to appear more concise because your text will be included in the tag itself. The other version of the `<c:out>` tag includes a body in order to allow a larger area of text to be displayed. This area can include multiple blank lines, text, and expressions.

Often, the `<c:out>` tag is used only to display the result of a simple expression. Listing 4.1 shows a simple use of the `<c:out>` tag.

Listing 4.1 Using the <c:out> Tag (out.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
  <head>
    <title>Out Examples</title>
  </head>

  <body>
    <h3>Out Example</h3>

    10 * 3 =
    <c:out value="\${10*3}" />

    <br />
  </body>
</html>
```

In Listing 4.1 you can see that the JSP is using the `<c:out>` tag to display the expression `\${10*3}`. The expression `\${10*3}` evaluates to 10 times 3, or 30. The expression is well defined and should always have a value. The `<c:out>` tag will display whatever the value attribute specifies.

Expressions will not always have values. An expression may default to null if there is insufficient information to calculate the expression. Consider the expression `\${a}`. This expression will display whatever is stored in the `a` variable. If the `a` variable has never been defined, then the `\${a}` expression is considered a null expression. The `<c:out>` tag

lets you specify a default value that will be displayed when the expression that you specified is null. Listing 4.2 shows the `<c:out>` tag used with a default value.

Listing 4.2 Using the `<c:out>` Tag with Defaults (`outdefault.jsp`)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
    <head>
        <title>Out Default Examples</title>
    </head>

    <body>testit =
        <c:out value="${testit}" default="Default Value" />
    </body>
</html>
```

As you can see, the code in Listing 4.2 attempts to display the value of the variable testit, which has not been defined and has a null value. However, the `<c:out>` tag specifies a default value. The `<c:out>` tag in Listing 4.2 will display the text Default Value. Without the default attribute, the code would have simply displayed a blank in place of the null value.

Working with null values in JSTL does not generally throw null exceptions as it does in regular Java. When a null value is accessed or compared, JSTL simply inserts an empty string in place of the null value. This prevents JSTL from throwing a `NullPointerException` as regular Java would.

As we saw earlier in this section, there are two forms of the `<c:out>` tag. The first form, which we have already examined, is a single tag that has no body. It is also possible to give a `<c:out>` tag a body. This body defines what will be output, just like the `value` attribute defined in the bodyless `<c:out>` tag.

By default, the `<c:out>` tag will automatically escape any special characters encountered. This can be good because it allows you to display special HTML reserved characters, such as greater than (`>`) and less than (`<`). When the `<c:out>` tag is presented with a less-than symbol, it will automatically replace the symbol with the HTML code `<`, which allows HTML to display the less-than symbol. Without this code, HTML would assume that your less-than symbol was the beginning of an HTML tag. Table 4.1 summarizes these special characters.

Table 4.1. Special Characters Encoded by `<c:out>`

Special Character	Encoded Form
<	<
>	>
&	&
'	'

"	"
---	--------

The automatic encoding of special characters by the `<c:out>` tag can also present a problem. You may want to store HTML tags, along with regular text, inside scoped variables. For example, consider the chat program discussed in [Chapter 3](#), "Understanding Basic Tag Logic." Each time the user enters some text, that text is added to an application variable. The following code is responsible for this task:

```
<c:set var="chat"
value="${chat}<b>${param.uid} :</b>${param.text}<br />"
scope="application" />
```

As you can see, this code adds to the already defined chat application variable. In addition to the text that the user has typed, the user's name should be added to the chat tag. The username is bolded before being added to the application variable.

When it comes time to display the chat application variable, it must be displayed without the special characters being encoded. If JSTL were to encode these special characters, the user would literally see `username` rather than just seeing *username* in bold characters. [Listing 4.3](#) shows how this encoding works.

Listing 4.3 Using <c:out> with/without Tag Escaping (outescape.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
  <head>
    <title>Out with Tag Escaping Examples</title>
  </head>

  <body>
    <c:set var="test" scope="page">
      <table border="0">
        <tr>
          <td bgcolor="red">&#160;</td>

          <td bgcolor="green">&#160;</td>
        </tr>

        <tr>
          <td bgcolor="blue">&#160;</td>

          <td bgcolor="yellow">&#160;</td>
        </tr>
      </table>
    </c:set>

    <h3>Out With Encode=true</h3>

    <c:out value="${test}" escapeXml="true" />
    <br />
```

```
<h3>Out With Encode=false</h3>

<c:out value="${test}" escapeXml="false" />

<br />
</body>
</html>
```

As you can see in [Listing 4.3](#), the page-scoped variable test is assigned some HTML that will display a multicolored table. The program then displays this variable twice.

The first time the variable is displayed with the `escapeXml` attribute set to true, which is the default. This is done using the following `<c:out>` tag:

```
<c:out value="${test}" escapeXml="true" />
```

Of course, because true is the default, the following line of code would do exactly the same thing:

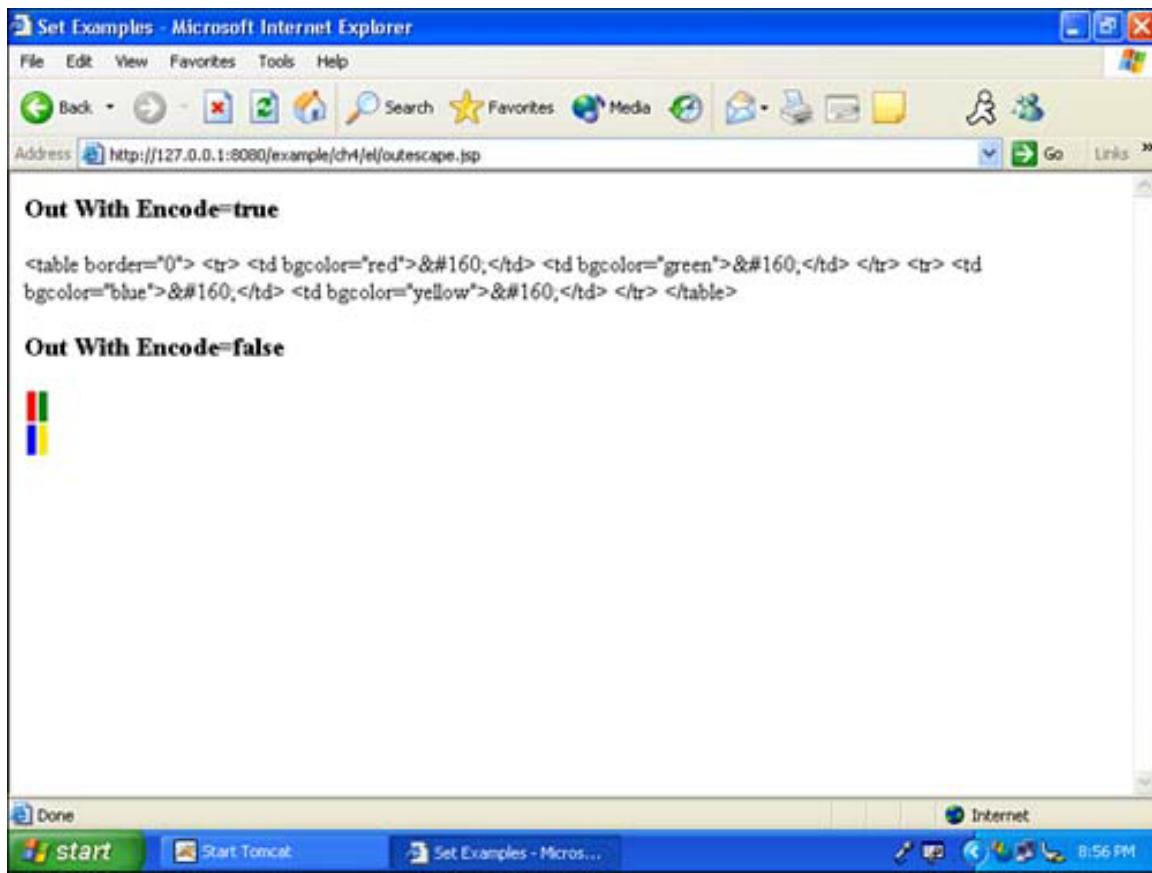
```
<c:out value="${test}" />
```

In this case, the colored HTML table will not be displayed; rather, the HTML that makes up the multicolored table will be displayed. Then, the program displays the multicolored table with the `escapeXml` attribute set to false, which displays the multicolored HTML table correctly. This is accomplished with the following `<c:out>` tag:

```
<c:out value="${test}" escapeXml="false" />
```

It is important to properly set this flag so that the program displays your variables correctly. [Figure 4.1](#) shows the output from this program.

Figure 4.1. HTML escaping and the `<c:out>` tag.



Using the <c:set> Tag

The `<c:set>` tag is used to set the value of a JSTL scoped variable. There are four forms of the `<c:set>` tag:

```
// Syntax 1: Set using value attribute
<c:set value="value"
var="varName" [scope="{page|request|session|application}"]/>
// Syntax 2: Set the value of a scoped variable using body content
<c:set var="varName" [scope="{page|request|session|application}"]>
body content
</c:set>
// Syntax 3: Set the value of a target object using value attribute
<c:set value="value"
target="target" property="propertyName"/>
// Syntax 4: Set the value of a target object using body text
<c:set target="target" property="propertyName">
body content
</c:set>
```

Just as with the `<c:out>` tag, the `<c:set>` tag lets you specify a value with the `value` attribute or use a tag body to specify the value. The bodyless version of `<c:set>` is generally used when only a small amount of text must be assigned to the variable. The body version of the `<c:set>` tag can be helpful when a large amount of text should be

assigned to the variable, specified by the var attribute of the `<c:set>` tag. Listing 4.4 shows how to use the `<c:set>` tag.

Listing 4.4 Using the `<c:set>` Tag (`set.jsp`)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
  <head>
    <title>Set Examples</title>
  </head>

  <body>
    <h3>Set With No Body</h3>

    <c:set var="str" value="Hello World" />

    str =
    <c:out value="${str}" />

    <br />

    <h3>Set With Body</h3>

    <c:set var="str">Hello Again, World</c:set>

    str =
    <c:out value="${str}" />

    <br />
  </body>
</html>
```

Listing 4.4 shows using the `<c:set>` tag both with and without a body. When used with a body, the `<c:set>` tag simply specifies both var, which specifies the variable to be set, and value, which specifies the value being set:

```
<c:out var="str" value="Hello world" />
```

When used with a body, this line becomes:

```
<c:set var="str">Hello World</c:set>
```

The `<c:set>` tag also allows you to specify scope. We discussed scope in [Chapter 2](#), "Programming the JSP Standard Tag Library."

It is also possible to set a property of an object using the `<c:set>` tag. Syntax 3 and 4 in our earlier example use this method. The following line of code shows how you would set the text property of an object named obj:

```
<c:out target="obj" property="text" value="Hello world" />
```

Scope is specified by using the scope attribute of the `<c:set>` tag. For example, the following set tag would be used to assign session scope:

```
<c:set var="test" value="Session Level Value" scope="session" />
```

Though not generally advised, it is possible to use the same variable name in two or more scopes. You could assign a different value to a given variable in the page, request, session, and application scopes. Listing 4.5 shows how this is done.

Listing 4.5 Using the <c:set> Tag with Scope (setscope.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
    <head>
        <title>Set in Scope Examples</title>
    </head>

    <body>
        <c:set var="test" value="Page Level Value" scope="page" />

        <c:set var="test" value="Request Level Value"
            scope="request" />

        <c:set var="test" value="Session Level Value"
            scope="session" />

        <c:set var="test" value="Application Level Value"
            scope="application" />

        <table border="1">
            <tr>
                <td>
                    <b>Default Level</b>
                </td>
                <td>
                    <c:out value="${test}" />
                </td>
            </tr>
            <tr>
                <td>
                    <b>Page Level</b>
                </td>
                <td>
                    <c:out value="${pageScope.test}" />
                </td>
            </tr>
            <tr>
                <td>
                    <b>Request Level</b>
                </td>
            </tr>
        </table>
    </body>
</html>
```

```

<td>
    <c:out value="${requestScope.test}" />
</td>
</tr>

<tr>
    <td>
        <b>Session Level</b>
    </td>

    <td>
        <c:out value="${sessionScope.test}" />
    </td>
</tr>

<tr>
    <td>
        <b>Application Level</b>
    </td>

    <td>
        <c:out value="${applicationScope.test}" />
    </td>
</tr>
</table>
</body>
</html>

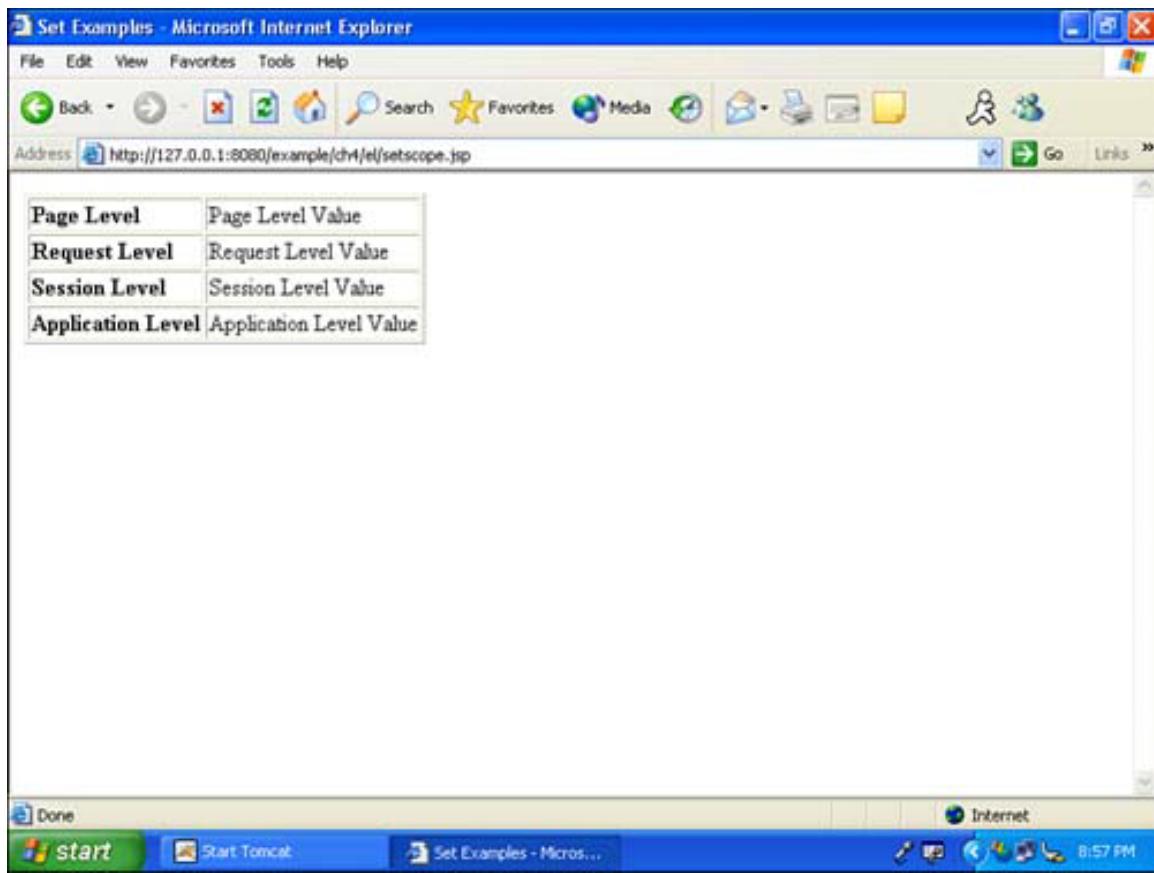
```

The code in [Listing 4.5](#) assigns a different value to the variable test in each of the scopes. When it's time to display test, you must tell JSTL which of the four scopes you would like displayed. You do this by prefixing the variable name with the scope name, followed by a period. For example, to specify that you would like to display the variable from the application scope, you would use the following line of code:

```
<c:out value="${applicationScope.test}" />
```

By specifying the scope, you are assured of getting the correct value. If you don't specify the scope, JSTL will begin looking for the variable in page scope. If it doesn't find your variable in page scope, JSTL then checks request, session, and finally application scope. The output from this program is shown in [Figure 4.2](#).

Figure 4.2. Variable scopes.



Using the <c:remove> Tag

The `<c:remove>` tag is a scoped variable. There is only one form of the `<c:remove>` tag:

```
<c:remove var="varName" [scope="{page|request|session|application}"] />
```

Removing a variable makes the memory space being used by that variable available for other variables that might be stored by your Web application. In actual programming practice, the `<c:remove>` tag is not used all that often. Page- and request-scoped variables are generally used for temporary variables, and they do not have to be removed manually—they will be automatically removed when the page exits. It is considered good programming practice to keep your session-scoped variables to a minimum, so temporary variables are rarely assigned to session scope. In addition, using temporary values in the application scope can cause threading problems. This is because many threads will be accessing your variable simultaneously. In any case, session-scoped variables are automatically removed when the session ends.

The `<c:remove>` tag can be useful when a value is assigned to application-level scope to be communicated to other sessions. In this case, the variable should be removed once it is no longer needed because it will not be removed automatically until either the Web server is shut down or the Web application is restarted.

Another somewhat common use of the `<c:remove>` tag is with session variables. Sometimes, it is necessary to take the user through a multiform input series. In such a situation, the user is presented with a series of forms that collects more and more information from the user. This data is not yet ready to be saved; it is not complete until the user reaches the final page. The data can be stored in a session variable that will be removed once the user either abandons the process or saves the form. Although this is a common technique, you should ensure that you do not store too much data in the user session due to the overhead incurred.

A technique for avoiding using session variables is to put intermediate data in hidden fields. This data will then be posted to subsequent forms and the program will still have access to it. This approach also protects the data from session expirations.

[Listing 4.6](#) shows an application that uses the `<c:remove>` tag.

Listing 4.6 Using the `<c:remove>` Tag (remove.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
  <head>
    <title>Remove Examples</title>
  </head>

  <body>
    <h3>Remove Example</h3>

    <c:set var="test" value="Hello World" scope="page" />

    The value in the variable test before remove is
    <c:out value="${test}" />

    <br />

    <c:remove var="test" scope="page" />

    The value in the variable test after remove is
    <c:out value="${test}" />

    <br />
  </body>
</html>
```

The program in [Listing 4.6](#) assigns the value Hello World to the variable test. The value stored in test is displayed to the browser. Then, the JSP page uses the `<c:remove>` tag to remove test. This is done with the following line:

```
<c:remove var="test" scope="page" />
```

This code will remove test. When test is redisplayed a few lines below the `<c:remove>` tag, a blank is displayed. This is because the removed test variable now has a null value.

JSTL always displays nulls as blanks. However, if you run `setscope.jsp` first, the result will be Session Level Value.

Using the EL Expression Language

EL provides two methods for using expressions. We discuss the first method, the EL expression language, next, and cover the RT expression language in the section that follows.

The designers of JSTL assumed that most developers will use the EL expression language. An expression using EL must always be delimited by \${ and }.

Accessing Properties

To access the properties of objects with EL, you should use either the . or [] operator. The . operator is used when you know ahead of time exactly what property you want to access. [Listing 4.7](#) shows both techniques.

Listing 4.7 Accessing Properties (prop.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
    <head>
        <title>Property Access</title>
    </head>

    <body>
        <c:if test="${pageContext.request.method=='POST'}">
            <c:set var="idx" value="name" />

            param.name =
            <c:out value="${param.name}" />

            <br />

            param[name] =
            <c:out value="${param[idx]}" />

            <br />
        </c:if>

        <br />

        <form method="post">Please enter your name?
        <input type="text" name="name" />

        <input type="Submit" />

        <br />
```

```
</form>
</body>
</html>
```

[Listing 4.7](#) begins by displaying the name property of the param object. The following line of code does this:

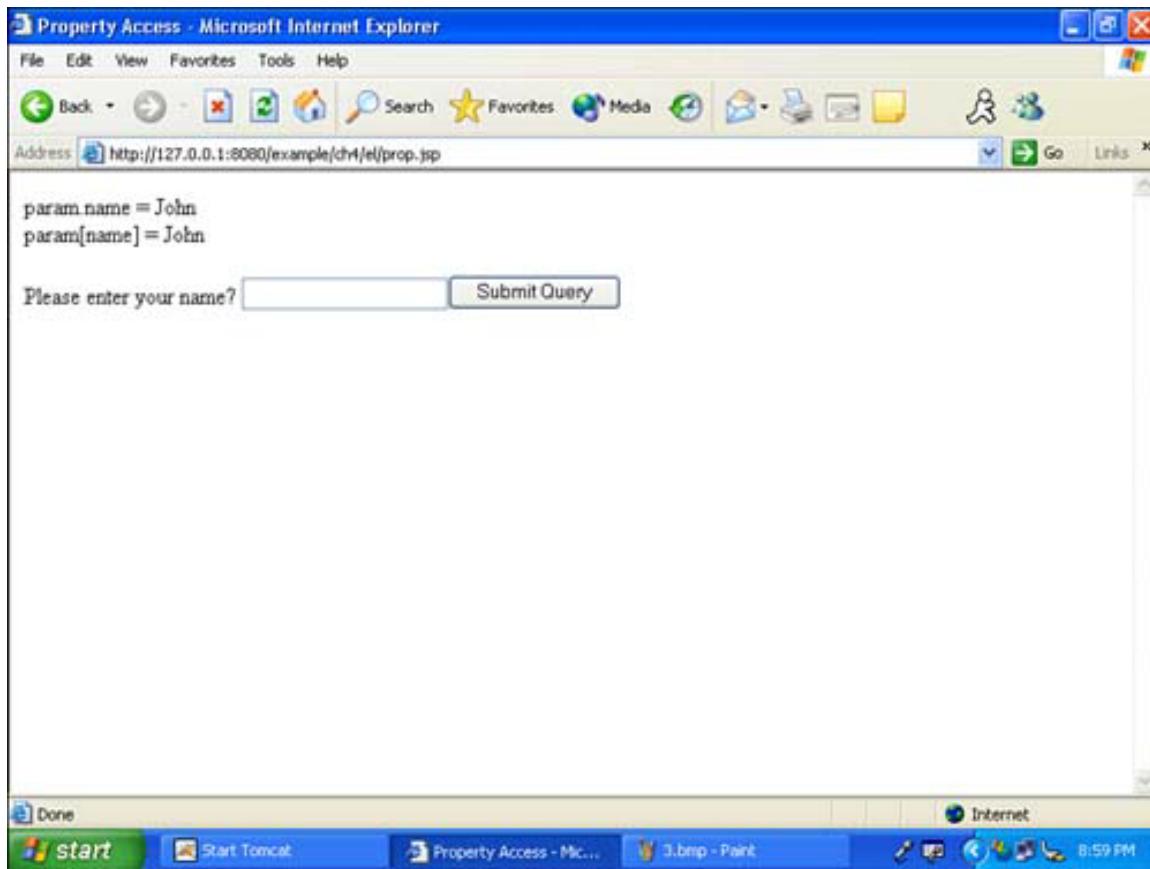
```
param.name = <c:out value="${param.name}" />
```

We have used this method of property access many times already. The only problem with this approach is that you cannot access a different property at runtime. To dynamically choose the property that you will access, you must use the [] operator. The following lines of code demonstrate this:

```
<c:set var="idx" value="name" />
param[name] = <c:out value="${param[idx]}" />
```

These lines also access the name property of the params collection. The difference is the variable idx is used to specify what property to use. [Figure 4.3](#) shows the output of this program.

Figure 4.3. Accessing properties.



Literals

Literals are constant values that you enter into EL code. EL supports four literal types:

Type	Values
Boolean	true and false.
Whole	Whole numbers such as 100, just like in Java.
Floating Point	Numbers such as 3.14. Unlike Java, JSTL allows you to use the d and f suffixes to differentiate between Double and Float.
	String-delineated with single or double quotes.
Null	The value null.

It is important to note that an EL expression cannot use the numeric suffixes d and f because they are used in Java to differentiate between Float and Double constants. The following JSP, which uses a `<c:set>` tag, can use the d suffix since the `<c:set>` tag is not being given an EL expression:

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

  <body>
    <c:set var="idx" value="3.140d" />

    idx = <c:out value="${idx}" /><br />

    idx = <c:out value="${idx == 3.14}" /><br />
  </body>
</html>
```

If this same value were tried with an EL expression, an error would result:

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

  <body>
    <c:set var="idx" value="3.140d" />

    idx = <c:out value="${idx}" /><br />

    idx = <c:out value="${idx == 3.14d}" /><br />
  </body>
</html>
```

Strings allow certain characters to be escaped within them. The quote ("") in a string is escaped as \". An apostrophe (') is escaped as \'. Finally, the backslash (\) is escaped as \\.

Property Access Operators: [] and .

EL expressions can access objects; therefore, it is necessary for EL expressions to be able to access the properties that are stored within the objects. EL provides the . and [] operators to do this.

Unlike in Java, the . and [] operators have the same meaning in EL. While the meaning of the operators are the same, they are used somewhat differently. The . operator works essentially the same as in Java; it is used to access an individual property in the object. For example, the following line of code would display the p property in the object obj. (In EL, a property can mean several things: It can be a public class scope variable, just as in Java, or it can also mean a getter method. For example, specifying obj.firstName would access the getter getFirstName.)

```
<c:out value="${obj.p}" />
```

This code allows you to access any property in an object or collection. The [] operator accesses the same property. However, you can put a variable inside the [] operator to allow you to specify which property you are seeking. For example, the following line of code would also display the p property in the object obj:

```
<c:out value="${obj['p']}" />
```

This may not seem any more useful than the previous line of code. However, this method becomes very powerful when you substitute another variable in the [] operator. Consider the following lines of code, which also display the p property of the object obj:

```
<c:set var="i" value="p"/>
<c:out value="${obj[i]}" />
```

Here, the variable i is used to specify the property that the program should retrieve. This allows your program to determine, at runtime, which property to use.

Arithmetic Operators

Arithmetic is provided to act on Integer (Long) and Floating Point. These operators will be discussed in this section.

Addition, Subtraction, and Multiplication

Addition, subtraction, and multiplication are handled by the typical +, -, and * operators. Given the equation of A{+,-,*}B, the following list summarizes the results of this operator:

- If A and B are null, then the result is 0.
- If A or B is Float or Double, then A and B are coerced to Double and the operator is applied.
- If A or B is Byte, Short, Character, Integer, or Long, then A and B are coerced to Long and the operator is applied.
- If the operator results in an exception, then an error is raised.
- Otherwise, an error is raised.

It is important to note that the EL plus operator cannot be used to concatenate two strings. If you did have two strings, str1 and str2, the following command would concatenate them into a third variable, named c:

```
<c:set var="c" value="${a}${b}" />
```

Division

Division is handled by the / operator. The textual operator div is also allowed for XPath compatibility. Like all EL operators, the rules for how the division is applied depends on the operands. Given the equation of A/B, the following list summarizes the results of this operator:

- If A and B are null, then the result is 0.
- Otherwise, coerce both A and B to Double and apply the operator.
- If division by zero, then return Infinity.
- If another exception, then raise an error.

The Modulo Operator

Modulo is handled by the % operator. The textual operator mod also allows for XPath compatibility. As you may recall, modulo is the remainder of a division. For example, 10%3 is 1, because 1 is the remainder of the division between 10 and 3. Like all EL operators, the rules for how the modulus is applied depends on the operands. Given the equation of A%B, the following list summarizes the results of this operator:

- If A and B are null, the result is 0.
- If A or B is Float or Double, coerce both to Double and apply the operator.
- Otherwise, coerce both A and B to Long and apply the operator. If the operator results in an exception, raise an error.

The Unary Minus Operator

The unary minus operator inverts the sign of a number. Placing the unary minus operator in front of a number—let's say 5—results in -5. Given the equation of -A, the following list summarizes the results of this operator:

- If A is null, the result is 0.

- If A is a string and unable to coerce to a Double, raise an error.
- If A is a string, coerce to a Double and apply the operator.
- If A is Byte, Short, Integer, Long, Float, or Double, then retain the type and apply the operator.
- If the operator results in an exception, raise an error.
- Otherwise, raise an error.

Relational Operators

The relational operators compare two variables to each other. These operators allow such comparisons as equal, not equal, and greater or less than. The relational operators are listed here:

- > and gt, greater than
- < and lt, less than
- <= and le, less than or equal
- >= and ge, greater than or equal

Given the equation of $A==B$, the following summarizes the results of these operators:

- If $A==B$, and the operator is $<=$ or $>=$, the result is true. Otherwise, the result is false.
- If A or B is Float or Double, coerce both A and B to Double and apply the operator.
- If A or B is Byte, Short, Character, Integer, or Long, then coerce both A and B to Long and apply the operator.
- If A or B is a string, then coerce both to a string and compare A and B lexically.
- If A is comparable, then if A.compareTo (B) throws an exception, an error is raised; otherwise, the result is returned.
- If B is comparable, then if B.compareTo (A) throws an exception, an error is raised; otherwise, the result is returned.
- Otherwise, raise an error.

Equality and Inequality Operators

Equality and inequality are handled by the $==$ and $!=$ operators. As with all EL operators, the rules for how this operator is applied depends on the operands. Given the equation $A==B$, the following list summarizes the results of this operator:

- If $A==B$, then the operator is applied.
- If A or B is null, then return false for $==$ or true for $!=$.
- If A or B is Float or Double, then coerce both A and B to Double and apply the operator.
- If A or B is Byte, Short, Character, Integer, or Long, then coerce both A and B to Long and apply the operator.

- If A or B is a Boolean, then coerce both A and B to a Boolean and apply the operator.
- If A or B is a string, then coerce both A and B to a string and compare lexically.
- Otherwise, if an error occurs while calling A.equals(B), then raise an error.
- Otherwise, apply the operator to the result of A.equals(B).

One unfortunate aspect of the == and != operators is that if a coercion fails, then an exception is thrown. This is somewhat counterintuitive. If the coercion fails, then you would assume that the values are in fact not equal.

Logical Operators

EL supports the usual complement of logical operators. These include and, or, and not. In this section, we examine these operators. The EL operators support the concept of "short-circuiting," just like the Java programming language. That means that an expression will stop being evaluated as soon as the result can be determined. For example, the and operator requires that both operands be true for the and operator itself to be true. If the first operand is evaluated to be false, then EL already knows that the expression can never be true. In this case, the and operator would return false without evaluating the expression any further.

The and and or Operators

The and and or operators are two basic logic operators. For an and expression to be true, both operands must be true. For an or to be true, either one or the other operator must be true. For an and to be true, both sides of the operation must be true.

- Attempt to coerce A and B to a Boolean; then return the result and apply the operator.
- If it is not possible to coerce A or B to a Boolean, then raise an error.

The Unary Not Operator

The unary not operator is handled by the ! and not operators. Given the equation of !A, the operation can be summarized as follows:

- Attempt to coerce A to a Boolean; then return the result and apply the operator.
- If it is not possible to coerce A to a Boolean, then raise an error.

Operator Precedence

The order of precedence is an important concept in any programming language. With only a few exceptions, the order of precedence in EL closely follows that of Java. The following list summarizes the order of precedence in EL:

1. [].

2. 0
3. - (unary) not ! empty
4. * / div % mod
5. + - (binary)
6. <> <= >= lt gt le ge
7. == != eq ne
8. && and
9. || or

Reserved Words

The following are the reserved words in EL, and may not be used as identifiers. We have already reviewed most of these.

NOTE

Many of the reserved words are not currently implemented in EL, but they may be used in the future. Sun has placed them in the reserved word list so they may serve as placeholders for future functionality.

- and
- div
- eq
- false
- ge
- gt
- instanceof
- le
- lt
- mod
- ne
- not
- null
- or
- true

Implicit Objects

EL provides several predefined objects, called implicit objects, which allow you to access application data that your JSP pages may need. [Table 4.2](#) lists these objects and their purposes.

<i>Table 4.2. EL Implicit Objects</i>
--

Implicit Object	Purpose
<code>pageContext</code>	Specifies the PageContext object.
<code>pageScope</code>	Provides access to all page-scoped objects.
<code>requestScope</code>	Provides access to all request-scoped variables.
<code>sessionScope</code>	Provides access to all session-scoped variables.
<code>applicationScope</code>	Provides access to all application-scoped variables.
<code>param</code>	Provides access to all single-value components that were posted from a form.
<code>paramValues</code>	Provides access to all array value components that were posted from a form.
<code>header</code>	Provides access to all single-value HTTP headers.
<code>headerValues</code>	Provides access to all array components that were sent as HTTP headers.
<code>cookie</code>	Provides access to the cookies.
<code>initParam</code>	Used to access XPath initialization parameters.

Type Conversion

When programming EL, you do not have direct control over types. When a variable is created in EL, the type is not specified. It is the context in which the variable is used that determines the type. For example, if you assign the value 10 to a variable and then print out that variable, the variable is coerced to be a string for display. If you then add 5 to that variable, the value of that variable will be coerced to a numeric type for the addition. This section summarizes the rules that EL uses in order to properly coerce one type into another.

Coerce to a String

Nearly any variable should be capable of being coerced to a string. The following summarizes this process when you attempt to coerce the variable A to a string and the situations when you might get an error:

- If A is a string, then the result is A.
- Otherwise, if A is null, then the result is "".
- Otherwise, if `A.toString()` throws an exception, then raise an error.
- Otherwise, the result is `A.toString()`.
- Otherwise, if A is a primitive type, return a string as in the Java expression ""+A.

Coerce to a Primitive Numeric Type

You may encounter numerous problems when you attempt to coerce a string to a primitive numeric variable. The following summarizes this process when you attempt to

coerce the variable A to a primitive numeric type and the situations when you might get an error:

- If A is null or "", then the result is 0.
- If A is a character, then convert to Short and apply the following rules:
 - If A is a Boolean, then raise an error.
 - If A is a number type N, then the result is A.
 - If A is a number with less precision than N, then coerce with no error.
 - If A is a number with greater precision than N, then coerce with no error.
 - If A is a string and `N.valueOf(A)` throws an exception, then raise an error.
 - If A is a string and `N.valueOf(A)` does not throw an exception, then return it as the result.
- Otherwise, raise an error.

Coerce to a Character

You may encounter numerous problems when you attempt to coerce to a primitive character as well. The following summarizes the process when you attempt to coerce the variable A to a character and the situations when you might get an error:

- If A is null or "", then the result is 0.
- If A is a character, then the result is A.
- If A is a Boolean, then raise an error.
- If A is a number with less precision than Short, then coerce with no error.
- If A is a number with greater precision than Short, then coerce with no error.
- If A is a string, then the result is `A.charAt(0)`.
- Otherwise, raise an error.

Coerce to a Boolean

There are a few times when you can coerce a type to a Boolean. The following summarizes this process when you attempt to coerce the variable A to a Boolean and the situations when you might get an error:

- If A is null or "", then the result is A.
- Otherwise, if A is a Boolean, then the result is A.
- Otherwise, if A is a string and `Boolean.valueOf(A)` throws an exception, then raise an error.
- Otherwise, if A is a string, then the result is `Boolean.valueOf(A)`.
- Coerce to any other type.
- Otherwise, raise an error.

Using the RT Expression Language

JSTL also includes support for scriptlet type expressions through the RT expression language. These expressions are bounded between the `<%=` and `%>` characters. By using the RT expression language, you can access familiar constructs from scriptlet programming.

JSTL implements support for two scripting languages through the use of twin tag libraries. To access the EL version of the if tag, the tag `<c:if>` is used. The RT version of the if tag can be accessed by using the `<c-rt:if>` tag. The four components of JSTL are each duplicated to support the RT expression language. [Table 4.3](#) shows four components that can use RT with JSTL. This is not defined in the JSTL 1.0 spec, but it is in the JSP 1.2 spec.

<i>Table 4.3. The Four Components of RT</i>		
Component	URI	Prefix
Core	http://java.sun.com/jstl/core-rt	c-rt
XML Processing	http://java.sun.com/jstl/xml-rt	x-rt
I18N Formatting	http://java.sun.com/jstl/fmt-rt	fmt-rt
Relational DB Access (SQL)	http://java.sun.com/jstl/sql-rt	sql-rt

For example, to take advantage of the core RT tags, you would use the following command:

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

WARNING

Of course, you must also make sure that your web.xml file has entries for the RT libraries if you want to use them. [Appendix B](#) covers this.

The RT expression language allows you to use expressions exactly as you did in JSP scriptlet programming. For example, the following `<c-rt:if>` tag checks the request method:

```
<c-rt:if test="<%request.getMethod().equals("POST") %>">
```

In general, you should always use an EL expression if one is available. However, there are a few cases where EL expressions do not provide the same functionality as RT. Let's examine a few of these cases.

Determining the Browser

Often, it is necessary to determine what browser the user is using. This can be done using some of the RT tags. [Listing 4.8](#) shows a brief example of how to determine the browser.

Listing 4.8 Determining the Browser (browser.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jstl/core-rt" prefix="c-rt" %>
<html>
    <head>
        <title>Check Browser</title>
    </head>

    <body>
        <c-rt:choose>
            <c-rt:when test="<%=request.getHeader(\"User-Agent\").
indexOf(\"MSIE\")!==-1%>">
                You are using Internet Explorer
            </c-rt:when>
            <c-rt:otherwise>
                You are using Netscape, or some other browser....
            </c-rt:otherwise>
        </c-rt:choose>
    </body>
</html>
```

As you can see in [Figure 4.4](#), the current browser is identified as Microsoft Internet Explorer. [Figure 4.5](#) shows the same program running with Netscape Communicator.

Figure 4.4. The current browser is Internet Explorer.

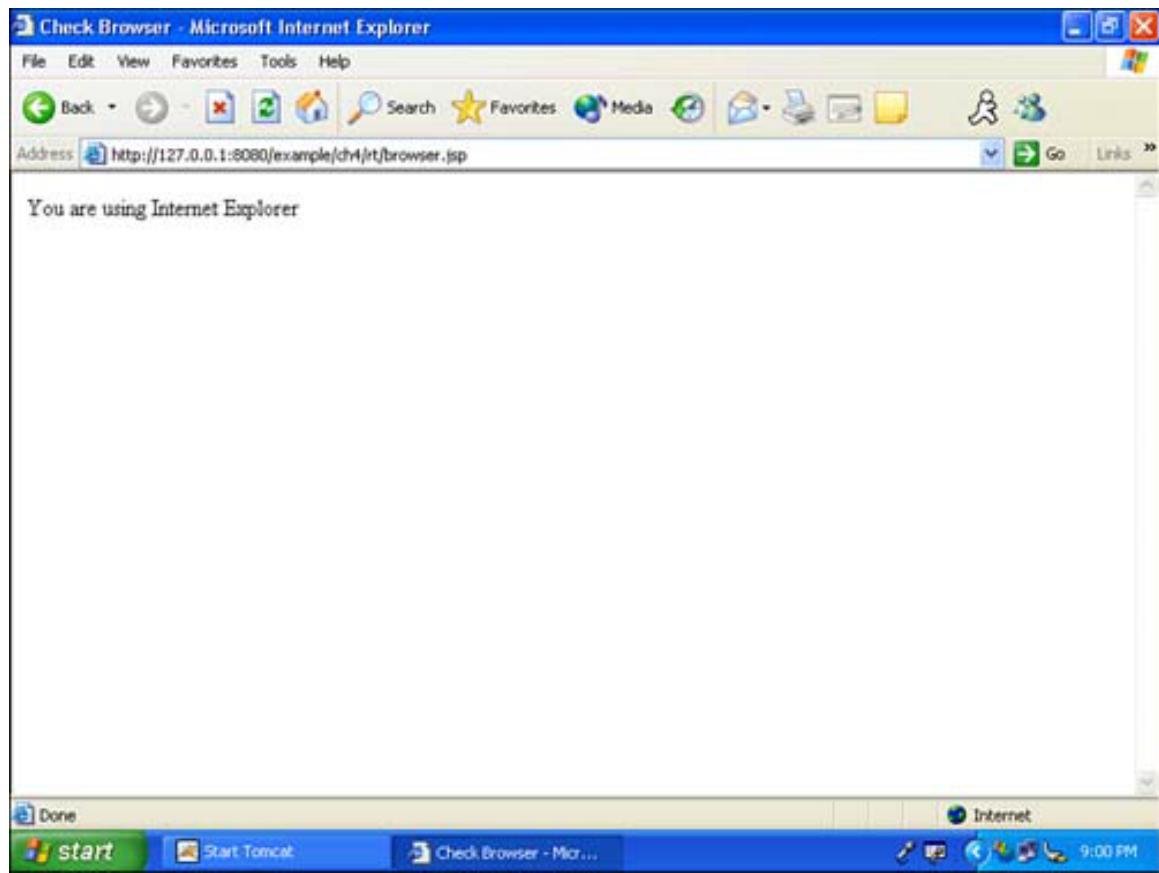
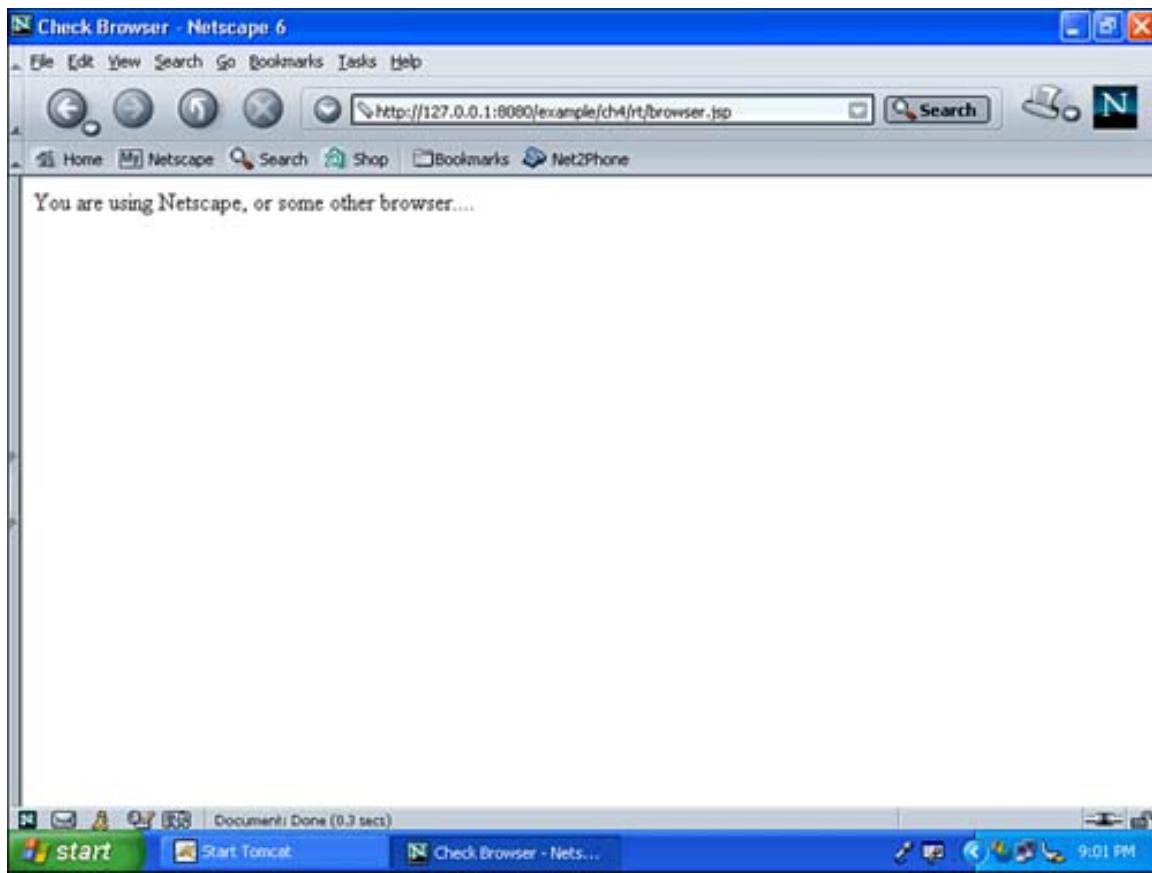


Figure 4.5. The current browser is Netscape Communicator.



In order to determine the current browser, the page must check the header variables. To do this, a `<c-rt:choose>` block is set up. Inside `<c-rt:when>`, test is used to retrieve the User-Agent, which specifies the browser being used. The following lines of code do this:

```
<c-rt:when test="<%request.getHeader(\"User-Agent\").indexOf(\"MSIE\") != -1%>">
    You are using Internet Explorer
</c-rt:when>
```

Case-Insensitive String Comparisons

The EL expression language does not have many commands for the manipulation of strings. When two strings are compared in EL, the comparison is always case sensitive. To do a comparison that is not case sensitive, it is necessary to use EL expressions.

[Listing 4.9](#) shows a simple JSTL program that uses RT expressions to do a case-insensitive comparison.

Listing 4.9 A Case-Insensitive String Comparison (if.jsp)

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

```

<title>If Caseless</title>
</head>

<body>
<c:set var="str" value="jStL" />

<jsp:useBean id="str" type="java.lang.String" />

<c-rt:if test="<%="str.equalsIgnoreCase(\"JSTL\") %>">
    They are equal
</c-rt:if>
</body>
</html>

```

To do a case-insensitive comparison, we must be able to use the `equalsIgnoreCase` found in the `String` class. In scriptlet JSTL, here's an expression that checks to see if the value stored in str is equal to JSTL (ignoring case):

```
<%=str.equalsIgnoreCase("JSTL") %>
```

This expression returns true if the compare matched. This expression can be directly included in a JSTL if statement. This requires an if statement from the RT tag library; the normal `<c:if>` tag that we've seen earlier will not work. To use an RT expression, you must use the `<c-rt:if>` tag. The complete if statement is shown here:

```

<jsp:useBean id="str" type="java.lang.String" />
<c-rt:if test="<%="str.equalsIgnoreCase(\"JSTL\") %>">
</c-rt:if>

```

RT scripts do not have direct access to JSTL scoped variables. To access such a variable, you have to map that variable into a bean first. This is the purpose of the `<jsp:useBean>` tag just before the if statement. The `<jsp:useBean>` tag shown here will make the scoped variable str available to RT expressions.

Summary

This chapter showed you how to use the expression language. The expression language allows you to modify scoped variables as it suits your program. JSTL can make use of both the new EL (to be included with JSP 1.3) and the older scriptlet-based expressions. The next chapter will explain how to use iterators so that you can work with larger sets of scoped variables.

Chapter 5. Collections, Loops, and Iterators

Collections have always been an important part of JSP programming. JSP pages often receive collections of objects, such as rows from a database that must be displayed to the HTML user. JSTL provides several tags that enable the JSP programmer to work with collections of objects. These tags allow a more consistent approach to handling collections than scriptlet JSP programming provides.

In addition to the ability to iterate over collections of objects, JSTL provides basic parsing into the iteration tags. Token iteration tags allow the programmer to iterate through strings that contain items separated by designated tokens. This allows a string to be tokenized and the results processed in a way that is appropriate for the page.

This chapter will introduce you to both iteration and tokenization. We begin by taking a closer look at the collections that can be used with these tags.

Understanding Collections

The term *collection* has a slightly different meaning in JSTL than it does in Java. In JSTL many different types of objects can function as a collection. In particular, the following types of classes often form collections in JSTL:

- Any class that implements the `java.util.Collection` interface
- Arrays of objects or primitive data types
- Any class that implements the `java.util.Iterator` interface
- Any class that implements the `java.util Enumeration` interface
- Any class that implements the `java.util.Map` interface
- Comma-delimited strings (`java.lang.String`)

All classes that are members of the Java collections API can be used with the JSTL iteration tags. These include `LinkedList`, `ArrayList`, `Vector`, `Stack`, `Set`, and `Map`. Additionally, string items that contain comma-delimited text can be used. Data is often collected using these collection types, and understanding how to access collections using JSTL is important.

One particularly handy feature of JSTL collections is that a comma-delimited string can be iterated over. For example, the list of values Monday, Tuesday, Wednesday, Thursday, Friday could be iterated over to return each day individually. Now, let's look at the JSTL tags that are used with JSTL collections.

The Iteration Tags

JSTL supports two tags that are used to iterate over collections. For iteration over the collections just discussed, you use the `<c:forEach>` tag. As we stated earlier, the `<c:forEach>` tag can iterate over comma-delimited strings. It is also possible to iterate over strings that are delineated by characters other than the comma. To iterate over such strings, you must use the `<c:forTokens>` tag. In this section, we discuss the `<c:forEach>` tag. In the next section, we will discuss the `<c:forTokens>` tag.

The `<c:forEach>` Tag

The `<c:forEach>` tag is used to iterate over a collection or to iterate a fixed number of times. There are two forms of the `<c:forEach>` tag:

```
// Syntax 1: Iterate over a collection of objects
<c:forEach [var="varName"] items="collection"
[varStatus="varStatusName"]
[begin="begin"] [end="end"] [step="step"]>
body content
</c:forEach>
// Syntax 2: Iterate a fixed number of times
<c:forEach [var="varName"]
[varStatus="varStatusName"]
begin="begin" end="end" [step="step"]>
body content
</c:forEach>
```

In syntax 1, you must provide the `<c:forEach>` tag with a collection to iterate over. You specify this collection by using the `items` attribute. Each item in the collection will be referenced by the variable specified in the `var` attribute of the `<c:forEach>` tag. This loop will continue until all of the items in the collection have been processed.

Syntax 2 does not rely on collections to set the boundaries that it will loop through. This example specifies a beginning, ending, and count for the loop. The attribute `begin` specifies where the loop should begin. The attribute `end` specifies where the loop should end; the `end` is inclusive. How far to move through each iteration of the loop is determined by the `step` attribute. Using these three attributes allows you to emulate the behavior of Java's `for` loop.

One deficiency of the fixed-loop `<c:forEach>` tag is that it will not loop backward. While it may seem logical that the following loop would count backward from 10, this loop would generate an exception:

```
<c:forEach begin="10" end="1" var="i" step="-1">
  <c:out value="${i}" /><br/>
<c:/forEach>
```

The `<c:forEach>` tag also includes the `varStatus` attribute, which enables you to obtain information about the iteration as it progresses. We discuss the `varStatus` attribute next.

WARNING

One limitation of the `<c:forEach>` tag is that the begin, end, and step attributes must be constant. It is invalid to specify another scoped variable for these values.

Loop Status

JSTL also gives you the ability to access information about the iteration as it proceeds. You accomplish this by using a varStatus variable. If you specify a varStatus variable, the iterator will fill it with a javax.servlet.jsp.jstl.core.LoopTagStatus object for each loop of the iterator. Because this is a standard Java object, you must use the RT expression language (explained in [Chapter 4](#), "Using the Expression Language") to access it.

One common use of this object is to get access to the iteration number we are on. We can then specify a different action for even and odd loop numbers. The following code shows how this is done:

```
<c:forEach var="product" items="${products}" varStatus="status">
    <jsp:useBean id="status"
        type="javax.servlet.jsp.jstl.core.LoopTagStatus"/>
    <c-rt:choose>
        <c-rt:when test="<%=>status.getCount() %2 ==0 %>">
            even item<br/>
        </c-rt:when>
        <c-rt:otherwise>
            odd item<br/>
        </c-rt:otherwise>
    </c-rt:choose>
</c:forEach>
```

As you can see, the varStatus object must be registered as a bean for it to be used. We do this with the following line:

```
<jsp:useBean id="status"
    type="javax.servlet.jsp.jstl.core.LoopTagStatus"/>
```

Registering this object allows you to access its properties. Our example uses the `getCount()` method. In addition to `getCount()`, you may wish to use some of the other methods we describe in the following sections.

The `getCurrent()` Method

The `getCurrent()` method has the following prototype:

```
public java.lang.Object getCurrent()
```

This method returns the current item from the iteration. Calling this method does not advance the iteration. Calling the method repeatedly returns the same object until the iteration is advanced.

The getIndex() Method

The `getIndex()` method has the following prototype:

```
public int getIndex()
```

This method returns the index of the current iteration. If the iteration is being performed over a subset of an underlying array, `java.lang.Collection`, or another type, the index returned is absolute. The index does not take into account the value of the begin, step, and end attributes. The index is that of the underlying Java collection and is 0-based.

The getCount() Method

The `getCount()` method has the following prototype:

```
public int getCount()
```

This method returns the count of the current round of the iteration. The count is a relative, 1-based sequence number that identifies the round of iteration. For example, an iteration with begin = 6, end = 10, and step = 2 produces the counts 1, 2, and 3, in that order.

The isFirst() Method

The `isFirst()` method has the following prototype:

```
public boolean isFirst()
```

This method returns true if this is the first round through the iteration. This method may return true even when the result of `getIndex()` is not equal to 0. This is because index specifies the absolute index of the current item based on the context of its underlying collection. However, it is always the case that a true result from `isFirst()` ensured that the value returned by `getCount()` will be 1. This difference in indexes is called *subsetting*.

The isLast() Method

The `isLast()` method has the following prototype:

```
public boolean isLast()
```

This method returns true if the iteration is in its last round. As is the case with `isFirst()`, subsetting is taken into account. (See the previous section for more information.)

The getBegin() Method

The `getBegin()` method has the following prototype:

```
public Integer getBegin()
```

This method returns the value of the begin attribute that was specified in the `<c:forEach>` or `<c:forTokens>` tag. If no begin tag was specified, null is returned.

The getEnd() Method

The `getEnd()` method has the following prototype:

```
public Integer getEnd()
```

This method returns the value of the end attribute that was specified in the `<c:forEach>` or `<c:forTokens>` tag. If no end tag was specified, null is returned.

The getStep() Method

The `getStep()` method has the following prototype:

```
public Integer getStep()
```

This method returns the value of the step attribute that was specified in the `<c:forEach>` or `<c:forTokens>` tag. If no step tag was specified, null is returned.

Using Iterators

Now that you have seen the `<c:forEach>` tag, let's look at a few examples of iterators. There are many common uses for iterators in JSP pages, and the following sections examine the most useful ones.

Iterating over Strings

As we mentioned earlier, the iterator tags can iterate over strings if the strings are comma delimited. Listing 5.1 shows a program that iterates over a string of comma-separated values.

Listing 5.1 Iteration and Strings (string.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
  <head>
    <title>String Collection Examples</title>
  </head>

  <body>
    <h3>String Collection Example</h3>
```

```

<c:set var="str"
  value="Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,
Saturday" />

<b>Input String:</b>
<br />
<br />
<c:out value="${str}" />
<br />
<br />
<b>Iterating:</b>
<br />
<br />
<c:forEach var="day" items="${str}">
  <c:out value="${day}" />
  <br />
</c:forEach>
</body>
</html>

```

In Listing 5.1, a string is assigned to hold the days of the week. Then a `<c:forEach>` tag is used to iterate through the list of string values. The following lines of code do this:

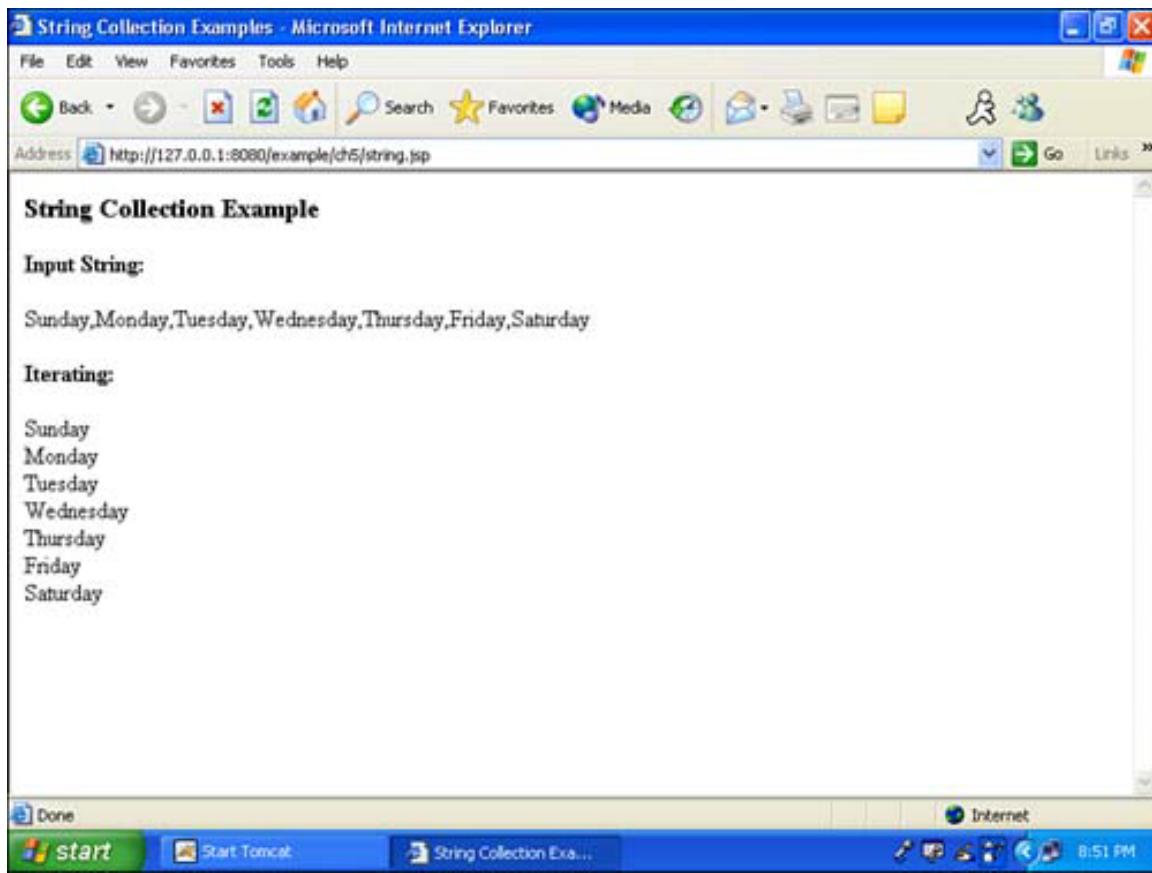
```

<c:forEach var="day" items="${str}">
  <c:out value="${day}" />
  <br />
</c:forEach>

```

As you can see, the string is sent to the tag as the `items` attribute, and each element of the string will be copied to the `day` scoped variable. This variable is then printed out through each round of the iteration. The output from this program is shown in Figure 5.1.

Figure 5.1. Iteration over a string.



Fixed-Length Loops

Collections are not the only sort of loop that can be done with the `<c:forEach>` tag. You may also use fixed loops that will count between a specified range of numbers. These loops resemble the for loops that the Java programming language provides. Listing 5.2 shows a program that makes use of fixed-length loops.

Listing 5.2 Fixed-Length Loops (fixed.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
    <head>
        <title>Count to 10 Example (using JSTL)</title>
    </head>

    <body>
        <table border="1">
            <tr>
                <td valign="top">
                    <h3>From 1 to 10</h3>

                    <c:forEach var="i" begin="1" end="10">
                        <c:out value="${i}" />

                        <br />
                </td>
            </tr>
        </table>
    </body>
</html>
```

```

        </c:forEach>
    </td>

    <td valign="top">
        <h3>From 10 to 1</h3>

        <c:forEach var="i" begin="1" end="10">
            <c:out value="${11-i}" />

            <br />
            </c:forEach>
        </td>
    </tr>

    <tr>
        <td>
            <h3>By Twos</h3>

            <c:forEach var="i" begin="2" end="10" step="2">
                <c:out value="${i}" />

                <br />
                </c:forEach>
            </td>

            <td valign="top">&ampnbsp;
            </td>
        </tr>
    </table>
</body>
</html>

```

The program in Listing 5.2 actually displays four types of loops. The first loop is just a simple count from 1 to 10, using the following lines of code:

```

<c:forEach var="i" begin="1" end="10">
    <c:out value="${i}" />
    <br />
</c:forEach>

```

This simple loop specifies a begin attribute of 1 and an end attribute of 10.

The second loop counts from 10 to 1 backward, using the following lines of code:

```

<c:forEach var="i" begin="1" end="10">
    <c:out value="${11-i}" />
    <br />
</c:forEach>

```

As we mentioned earlier, the iteration tags are not capable of counting backward. Because of this, we use nearly the same `<c:for>` tag as the count-to-10 example. The only difference is that the out tag uses the expression \${11-i} to adjust the output from this loop to appear as if the program is counting backward from 10.

The next loop shows how to use the step attribute. If a step attribute is specified, the count can skip numbers. The following lines of code count from 2 to 10 by twos:

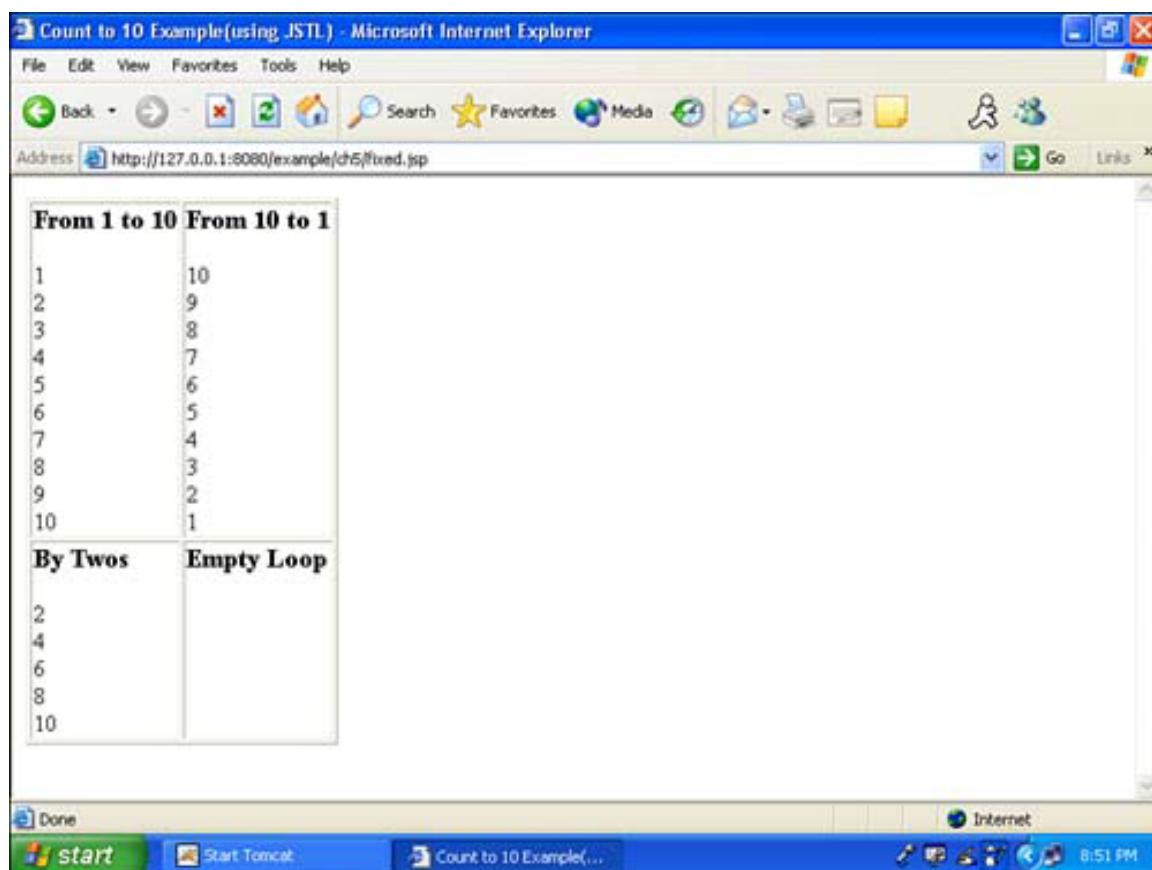
```
<c:forEach var="i" begin="2" end="10" step="2">
    <c:out value="${i}" />
    <br />
</c:forEach>
```

Keep in mind that the beginning index must not be greater than the ending index. This will cause an exception to be raised, as shown here:

```
<c:forEach var="i" begin="101" end="100" step="500">
    <c:out value="${i}" />
</c:forEach>
```

This program shows three different ways the `<c:forEach>` tag can be used to create fixed strings. The program's output is shown in [Figure 5.2](#).

Figure 5.2. Fixed-length loops.



The <c:forTokens> Tag

The `<c:forEach>` tag has the ability to iterate over a string of comma-delineated items. JSTL also provides the more general `<c:forTokens>` tag, which allows you to iterate over a string that is delimited by values other than commas. Here is one form of the `<c:forTokens>` tag:

```
<c:forTokens items="stringOfTokens" delims="delimiters"
[var="varName"]
[varStatus="varStatusName"]
[begin="begin"] [end="end"] [step="step"]>
body content
</c:forEach>
```

The `<c:forTokens>` tag's syntax closely resembles that of the `<c:forEach>` tag. The attribute `items` is used to pass in the string to be parsed. The attributes `begin` and `end` allow you to specify beginning and ending items, which means you can make the iteration skip elements from the beginning or end of the list. The `step` attribute allows you to skip elements within the list—for example, a `step` attribute with the value of 2 would process only every other item.

The `delims` attribute allows you to specify what delimiters to use. JSTL can parse only single-character delimiters. If you specify more than one character as the delimiter, JSTL will end an item on any of the characters that you provided as a delimiter. For example, the delimiter string,`,;:` would break up items based on the occurrence of the comma(`,`), semicolon(`;`), or colon(`:`).

The `<c:forTokens>` tag also includes the `varStatus` attribute. This attribute enables you to obtain information about the iteration as it progresses.

Using the `<c:forTokens>` tag, you can easily process delimiters other than the comma. Listing 5.3 shows a program that will separate a sentence into its component words.

Listing 5.3 Iterating over Tokens (token.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
  <head>
    <title>Updatable Collections</title>
  </head>

  <body>
    <table border="0">
      <form method="post">
        <tr bgcolor="blue">
          <td>
            <p align="center">
              <b>
                <font color="#FFFFFF">Parse for Tokens</font>
              </b>
            </p>
          </td>
        </tr>
      </form>
    </table>
  </body>
</html>
```

```

        </p>
        </td>
    </tr>

    <tr>
        <td valign="top">
            <p align="left">Enter a sentence:
            <br />

            <input width="20" maxwidth="20" name="text"
size="50" />

            <br />

            &#160;</p>
        </td>
    </tr>

    <tr>
        <td valign="top">
            <p align="center">
                <input type="submit" name="parse" value="Parse" />
            </p>
        </td>
    </tr>
</form>
</table>

<c:if test="${pageContext.request.method=='POST' }">
    <table border="1">
        <c:set var="i" value="1" />
        <c:forTokens items="${param.text}" var="word"
delims=" ,.?!">
            <c:set var="i" value="${i+1}" />

            <tr>
                <td>
                    <b>Word
                    <c:out value="${i}" />
                    </b>
                </td>

                <td>
                    <c:out value="${word}" />
                </td>
            </tr>
        </c:forTokens>
    </table>
</c:if>
</body>
</html>

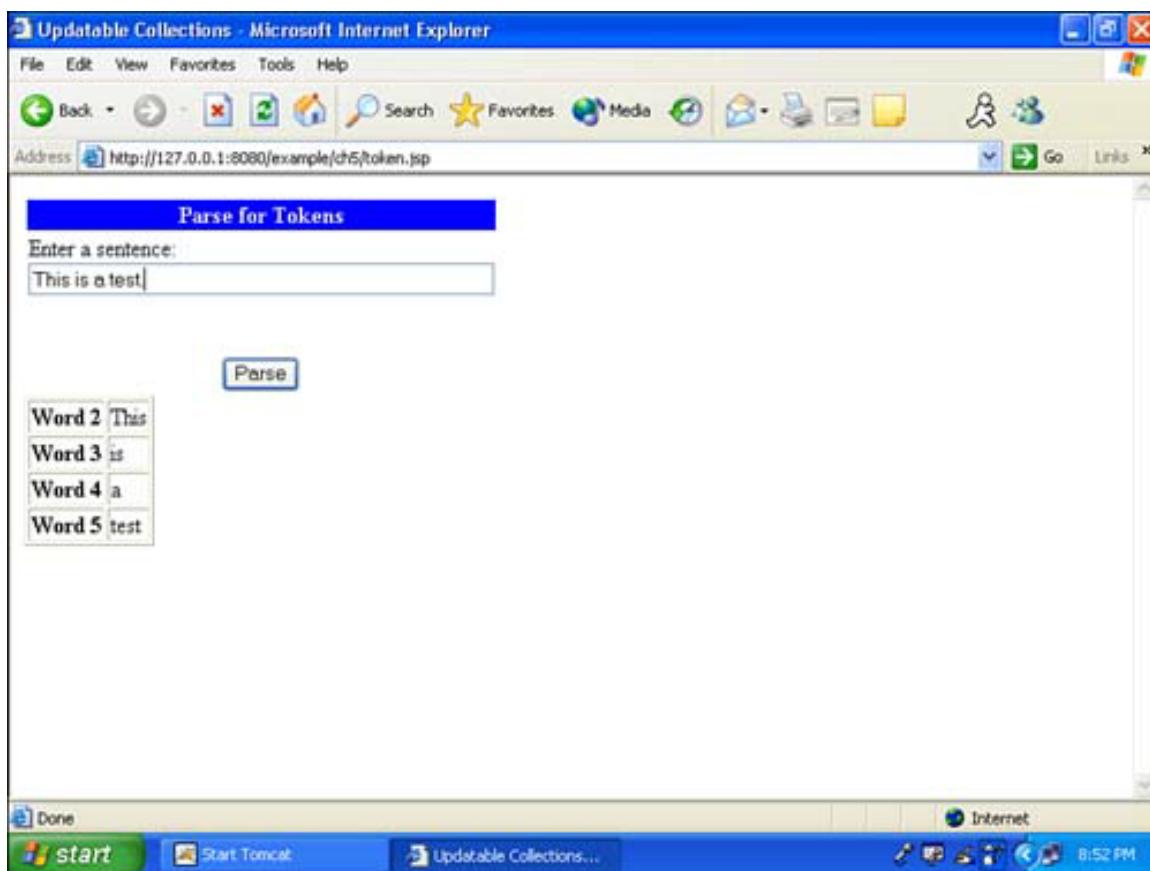
```

The program in Listing 5.3 begins with a typical HTML form that asks the user to input a sentence. When the user does so, the sentence is broken down into its component words and displayed to the user. Our program uses the `<c:forTokens>` tag to break up this string, as shown here:

```
<c:forTokens items="${param.text}" var="word" delims=" ,.?!\'">
```

As you can see, this code specifies several delimiters. Most of the normal English punctuation characters are used as delimiters, and thus would count as an end to a word. As our program loops through the string, each word is printed out. The program also maintains a counter variable, i, to display the current position in the loop. (You could also keep count by using a varStatus attribute. We discuss the varStatus attribute again in the section "Tag Collaboration with a Fixed Loop" section later in this chapter.) The output from this program is shown in [Figure 5.3](#).

Figure 5.3. Iterating over tokens.



Modifying a Collection

So far, the collections that we have examined have remained static. In this section, we show you how to create a comma-delineated collection that can have elements dynamically added and removed from it. [Listing 5.4](#) shows this program.

Listing 5.4 Modifying a Collection (modify.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<c:if test="${pageContext.request.method=='POST'}">
```

```

<c:choose>
    <c:when test="${param.add!=null}">
        <c:if test="${sessionScope.list!=null and sessionScope.list!=''}">
            <c:set var="list" value="${list}, ${param.item}" scope="session" />
        </c:if>

        <c:set var="list" value="${list}${param.item}" scope="session" />
    </c:when>

    <c:when test="${param.remove!=null}">
        <c:set var="list2" value="" />

        <c:forEach var="item" items="${sessionScope.list}">
            <c:if test="${item!=param.item}">
                <c:if test="${list2!=''}">
                    <c:set var="list2" value="${list2}, ${item}" />
                </c:if>

                <c:set var="list2" value="${list2}${item}" />
            </c:if>
        </c:forEach>

        <c:set var="list" value="${list2}" scope="session" />

        <c:remove var="list2" />
    </c:when>
</c:choose>
</c:if>

<html>
    <head>
        <title>Updatable Collections</title>
    </head>

    <body>
        <table border="0">
            <form method="post">
                <tr bgcolor="blue">
                    <td colspan="2">
                        <font color="white">Updatable Collections</font>
                    </td>
                </tr>

                <tr>
                    <td valign="top">
                        <select NAME="choice" SIZE="5" width="20">
                            <c:forEach var="item" items="${sessionScope.list}">
                                <option>
                                    <c:out value="${item}" />
                                </option>
                            </c:forEach>
                        </select>
                    </td>
                </tr>
            </table>
        </body>
    </html>

```

```

<br />

<input width="20" maxwidth="20" name="item" size="20" />

<br />

<input type="submit" name="add" value="Add" />

<input type="submit" name="remove" value="Remove" />
</td>
</tr>
</form>
</table>
</body>
</html>

```

The program in Listing 5.4 works by displaying a list box next to a text field and two buttons. The user enters a value in the text field, and uses the buttons to either add or remove that item from the list box.

First, let's explain how to add a string to the list box. The contents of the list box are stored as a comma-delimited list in a session variable named list. When the first item is to be added to the list box, the list session variable is examined to see if there are any current values. If values already exist, the program appends a comma to the end of the list. After this, the new item that the user entered is appended to the list. The following lines of code do this:

```

<c:if test="${sessionScope.list!=null and sessionScope.list!=''}">
  <c:set var="list" value="${list}, ${param.item}" scope="session" />
</c:if>
<c:set var="list" value="${list}${param.item}" scope="session" />

```

The process of removing an item from the list is more complex. It is possible that the item to be removed might exist anywhere in the comma-separated list. To remove this item, the program must rebuild the list line by line. First, a page-scoped variable named list2 is initialized to an empty string. This variable will receive the contents of the new list:

```
<c:set var="list2" value="" />
```

Next, the program must iterate through the current list. The following `<c:forEach>` tag sets up to iterate through the current list:

```
<c:forEach var="item" items="${sessionScope.list}">
```

As items are encountered in the list, they are added to the newly created list in the variable list2. However, we must filter out the item that we want removed. If we fail to do this, we will simply re-create the list with the item still in the list. This is done with the following line of code:

```
<c:if test="${item!=param.item}">
```

Once we have verified that the current item is not the one that is to be removed, we must add it to the list. First, we must see if the new list is empty—if it is, then a comma will be appended to the list. This prevents the new list from starting with a comma rather than the first value. The following line of code makes this test and adds the comma if it is required:

```
<c:if test="${list2!=''}">
```

If the program finds that the list is not empty, then it adds a comma to separate the new item from what is already in the list:

```
<c:set var="list2" value="${list2}, " />
</c:if>
```

Now that the comma has been added to the list, we can add the new item. The following lines of code complete this process:

```
<c:set var="list2" value="${list2}${item}" />
</c:if>
</c:forEach>
```

At this point, the list has been re-created in the new list2 variable. The next step is to copy the new variable back to the old session list. The following line of code performs this step:

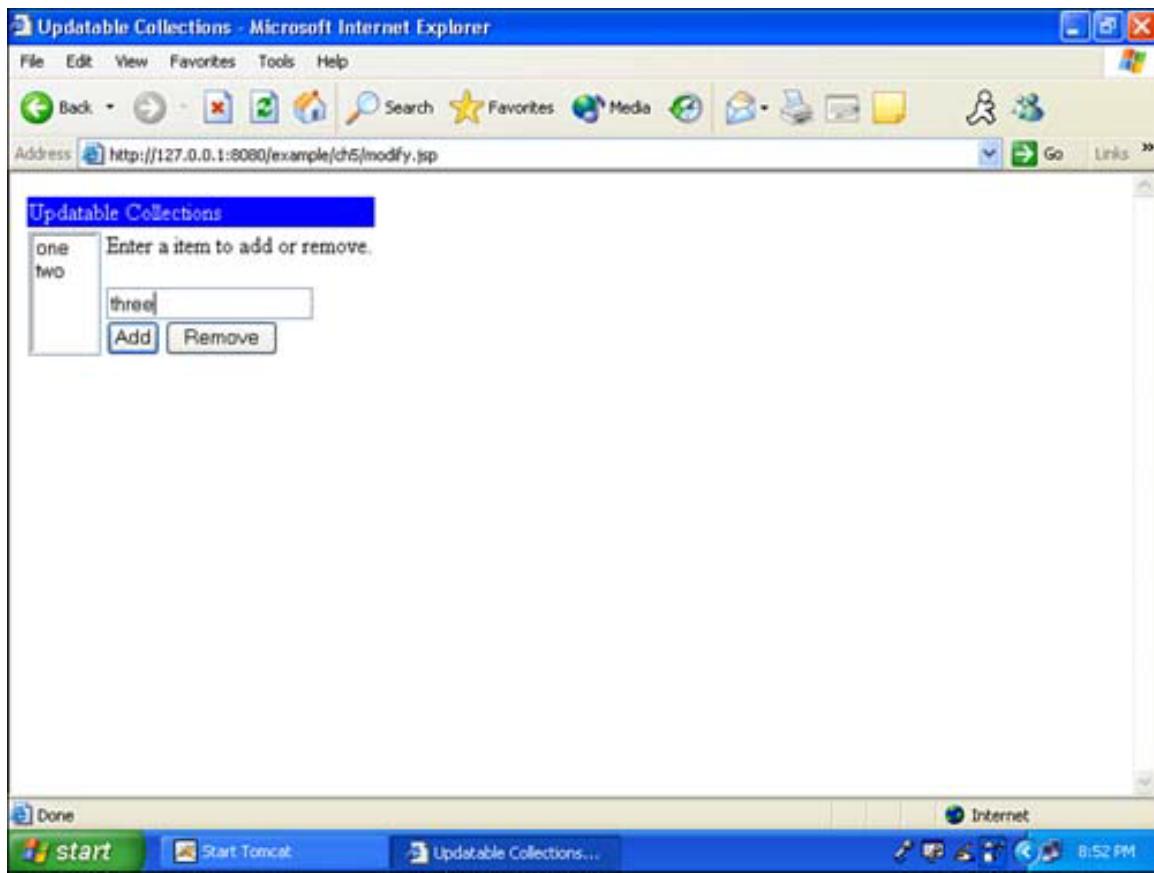
```
<c:set var="list" value="${list2}" scope="session" />
```

Now that the new list has been copied, the temporary list, stored in list2, can be deleted. This is done with the following `<c:remove>` tag:

```
<c:remove var="list2" />
```

This program demonstrates most of the concepts of iteration. The output can be seen in [Figure 5.4](#).

Figure 5.4. Modifying a collection.



Tag Collaboration with a Fixed Loop

Displaying lists is common in Web applications. Often, these lists are displayed in alternating colors. This resembles the lines of a ledger, such as one for a checking account.

Reproducing this ledger look in a Web application is a good use for the `varStatus` attribute of both the `<c:forEach>` and the `<c:forTokens>` tags. By using the `getCount()` method of the `varStatus` variable, you can determine whether you are on an even or odd row. Your program can then display the appropriate color because it alternates the background color for even and odd rows.

Let's look at an example. Listing 5.5 shows how the `varStatus` attribute can be used to alternate colors on a fixed-size loop.

Listing 5.5 Tag Collaboration (evenodd.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jstl/core-rt" prefix="c-rt" %>
<html>
    <head>
        <title>Count to 10 Example(tracking even and odd)</title>
    </head>
```

```

<body>
    <table border="0">
        <c:forEach var="i" begin="1" end="10" varStatus="status">
            <jsp:useBean id="status"
                type="javax.servlet.jsp.jstl.core.LoopTagStatus" />

            <c:rt:choose>
                <c:rt:when test="<%==status.getCount()%2==0%">
                    <c:set var="color" value="#eeeeee" />
                </c:rt:when>

                <c:rt:otherwise>
                    <c:set var="color" value="#dddddd" />
                </c:rt:otherwise>
            </c:rt:choose>

            <tr>
                <td width="200" bgcolor=<c:out value="${color}" />>
                <c:out value="${i}" />
            </td>
        </tr>
    </c:forEach>
</table>
</body>
</html>

```

In Listing 5.5, the program's loop is simple and is identical in concept to the count-to-10 example. The primary difference is that we use the varStatus attribute to gain access to the row counter. As shown here, this loop starts off just like the count-to-10 example:

```
<c:forEach var="i" begin="1" end="10" varStatus="status">
```

The primary difference in this loop is that the scoped variable status is specified in the varStatus attribute. This will make the status information available as the loop processes.

To take advantage of the status variable, we must map that variable to a bean. This is because the status object returns most of its information through method calls. The EL expression language is not capable of receiving information this way. Therefore, we'll use the RT expression language. As explained in Chapter 4, when using the RT expression language, all objects that are to be used must be mapped to beans. The following code maps the status variable to a bean:

```
<jsp:useBean id="status"
    type="javax.servlet.jsp.jstl.core.LoopTagStatus" />
```

Now that the RT expression language has access to the status object, we must check the count. This is done by using a `<c-rt:choose>` tag. Here, we can compare the count and take two alternating actions, depending on whether the current row is even or odd. The following code checks to see whether the row is even:

```
<c-rt:choose>
```

```
<c-rt:when test="<%==status.getCount()%2==0%>">
    <c:set var="color" value="#eeeeee" />
</c-rt:when>
```

If the row is found to be even, then a color is assigned to the page-scoped variable named color. To handle odd values, we use this code:

```
<c-rt:otherwise>
    <c:set var="color" value="#dddddd" />
</c-rt:otherwise>
</c-rt:choose>
```

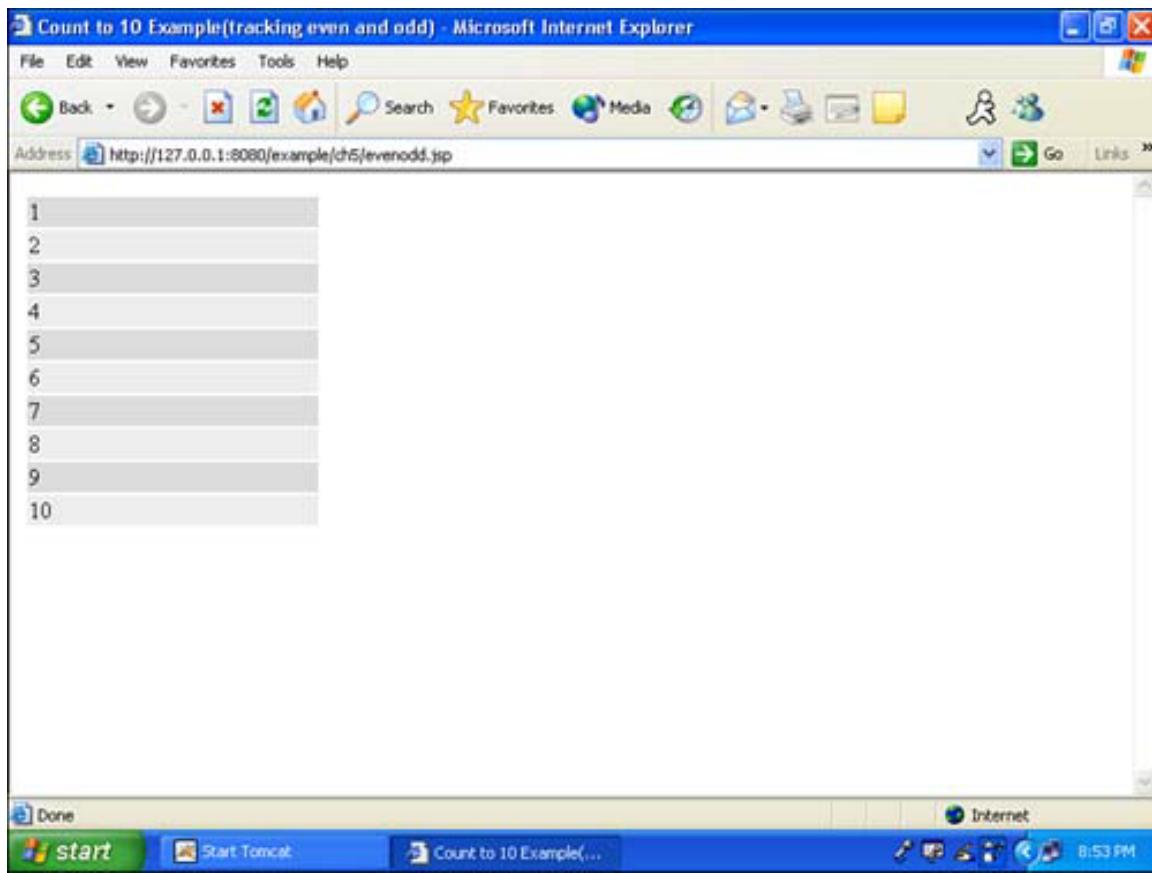
For odd values, a different color is saved to the color page-scoped variable.

Now that our program has determined the color, it must use that color in the HTML. The following code causes the color to appear as the background color of table rows:

```
<tr>
    <td width="200" bgcolor=<c:out value="${color}" />>
        <c:out value="${i}" />
    </td>
```

Figure 5.5 shows the output from this program.

Figure 5.5. Tag collaboration.



Tag Collaboration with a Collection

The varStatus attribute can be used with collections as well as fixed-loop tags. Both the `<c:forEach>` and the `<c:forTokens>` tags support the use of a varStatus tag. Listing 5.6 shows a collections-based program that makes use of the varStatus attribute as well.

Listing 5.6 Collections-Based Tag Collaboration (evenoddlist.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jstl/core-rt" prefix="c-rt" %>
<html>
  <head>
    <title>Count to 10 Example(tracking even and odd)</title>
  </head>

  <body>
    <c:set var="days"
      value="Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday" />

    <table border="0">
      <c:forEach var="i" items="${days}" varStatus="status">
        <jsp:useBean id="status"
          type="javax.servlet.jsp.jstl.core.LoopTagStatus" />

        <c-rt:choose>
```

```

<c-rt:when test="<%==status.getCount()%2==0%">
    <c:set var="color" value="#eeeeee" />
</c-rt:when>

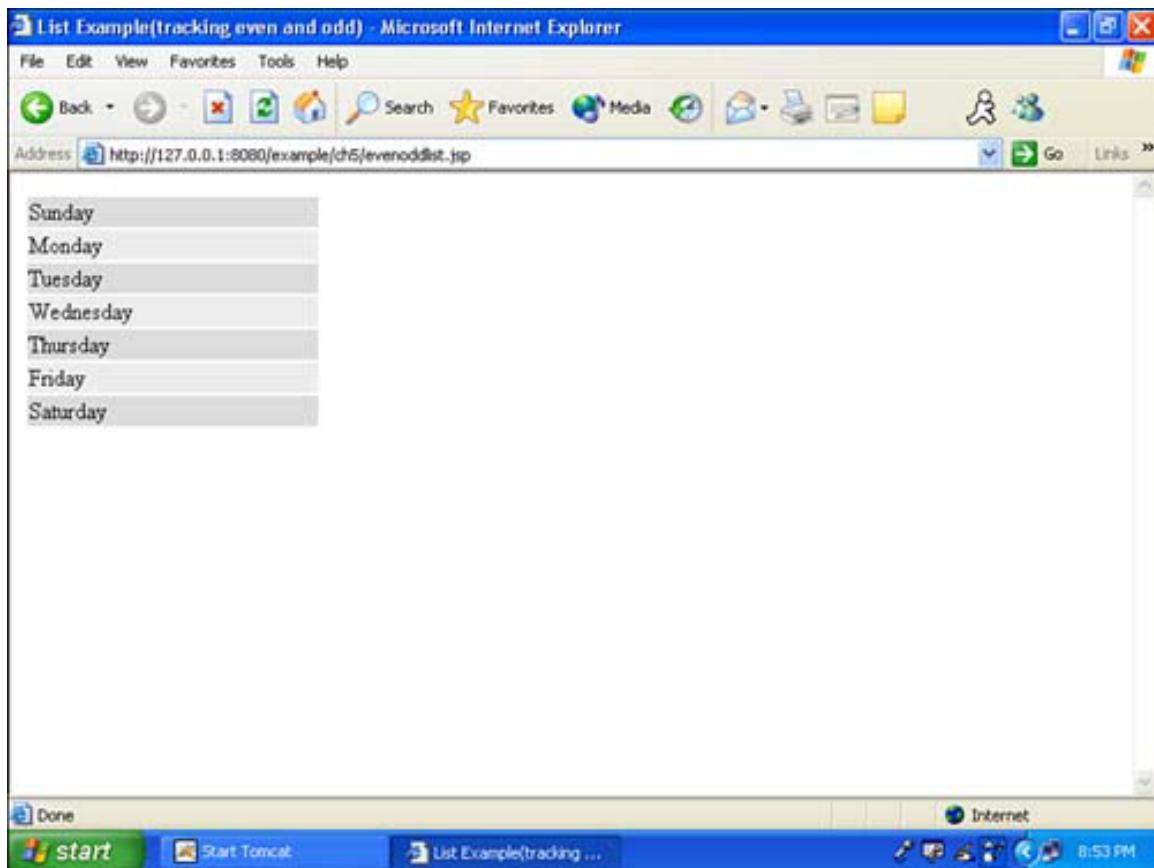
<c-rt:otherwise>
    <c:set var="color" value="#dddddd" />
</c-rt:otherwise>
</c-rt:choose>

<tr>
    <td width="200" bgcolor=<c:out value="${color}" />>
    <c:out value="${i}" />
    </td>
</tr>
</c:forEach>
</table>
</body>
</html>

```

As you can see in Listing 5.6, the basic loop construct is the same as in our previous example. This makes the `<c:forEach>` and `<c:forTokens>` tags interchangeable. Figure 5.6 shows the output from this program.

Figure 5.6. Tag collaboration with a collection.



Summary

In this chapter, we showed you how to use iterators. Iteration is one of the most important concepts in Web programming. As you progress through the remaining chapters in this book, you will use iterations to process other types of data, such as the results from an SQL database.

This chapter also showed that you can use iterators to perform basic string processing. In the next chapter, you will learn how to properly format data for display using JSTL's formatting tag library.

Chapter 6. Formatting Data with Tags

JSTL provides several tags that greatly simplify the formatting and parsing of data. Tags are provided that allow you to format numbers, percents, dates, and currencies. By using these tags, you can fine-tune the output of your data, without the need for customized display programming.

Web applications must frequently process data entered by a user. Data from a user always arrives as a string. However, this string is often a number, date, percentage, or other data type. JSTL provides parsing tags that enable you to take the strings a user enters and parse them into other data types.

In this chapter, we examine both of these issues. We provide several examples that illustrate practical application of data formatting and parsing. We begin by examining the process of data formatting.

Formatting

Formatting and parsing, as defined by JSTL, are two related but quite different actions. JSTL considers formatting to be the process of changing the appearance of stored data. This data has already been validated to be correct, and most likely should not produce an error.

JSTL provides two tags for data formatting. The `<fmt:formatNumber>` tag is used to format numbers; the `<fmt:formatDate>` tag is used to format dates and times.

Using the `<fmt:formatNumber>` Tag

The `<fmt:formatNumber>` tag is used to format numbers, percentages, and currencies. There are two forms of this tag:

```
// Syntax 1: Without a body
<fmt:formatNumber value="numericValue"
  [type="{number|currency|percent}"]
  [pattern="customPattern"]
  [currencyCode="currencyCode"]
  [currencySymbol="currencySymbol"]
  [groupingUsed="{true|false}"]
  [maxIntegerDigits="maxIntegerDigits"]
  [minIntegerDigits="minIntegerDigits"]
  [maxFractionDigits="maxFractionDigits"]
  [minFractionDigits="minFractionDigits"]
  [var="varName"]
  [scope="{page|request|session|application}"]/>
// Syntax 2: With a body to specify the numeric value to be formatted
<fmt:formatNumber [type="{number|currency|percent}"]>
```

```

[pattern="customPattern"]
[currencyCode="currencyCode"]
[currencySymbol="currencySymbol"]
[groupingUsed="{true|false}"]
[maxIntegerDigits="maxIntegerDigits"]
[minIntegerDigits="minIntegerDigits"]
[maxFractionDigits="maxFractionDigits"]
[minFractionDigits="minFractionDigits"]
[var="varName"]
[scope="{page|request|session|application}"]>
    numeric value to be formatted
</fmt:formatNumber>

```

The attributes accepted by the `<fmt:formatNumber>` tag include the following:

Attribute	Required	Purpose
<code>currencyCode</code>	N	String ISO 4217 currency code. Applied only when formatting currencies (that is, if type is equal to currency); ignored otherwise.
<code>currencySymbol</code>	N	String currency symbol. Applied only when formatting currencies (that is, if type is equal to currency); ignored otherwise.
<code>groupingUsed</code>	N	Specifies whether the formatted output will contain any grouping separators. The default value is true.
<code>maxFractionDigits</code>	N	The maximum number of digits in the fractional portion of the formatted output.
<code>maxIntegerDigits</code>	N	The maximum number of digits in the integer portion of the formatted output.
<code>minFractionDigits</code>	N	The minimum number of digits in the fractional portion of the formatted output.
<code>minIntegerDigits</code>	N	The minimum number of digits in the integer portion of the formatted output.
<code>pattern</code>	Y	The string custom formatting pattern. Applied only when formatting numbers (that is, if type is missing or is equal to number); ignored otherwise.
<code>scope</code>	N	The scope of var. The default value is page.
<code>type</code>	N	Specifies whether the value is to be formatted as a number, currency, or percentage. The default value is number.
<code>value</code>	N	The string or number numeric value to be formatted.
<code>var</code>	N	The name of the exported scoped attribute that stores the formatted result as a string.

In the first syntax, the value to be formatted is passed in using the value attribute. The formatted number will be written to the page if a var attribute is not specified. If a var attribute is specified, it will receive the formatted number as a string.

The second syntax uses the body of the tag to specify the number that is to be formatted. You should not provide a value attribute if you are using the tag with a body (syntax 2, in our example). In addition to these standard JSTL attributes, several formatting attributes are available specific to number formatting.

A type attribute is provided to specify what type of number this is. Valid values for the type attribute are number, currency, and percent.

If the type attribute is currency, then the currencyCode and currencySymbol attributes can also be specified. The currencyCode attribute specifies an ISO4217 currency code for this number. This allows JSTL to know what type of currency you are providing. You can also override the default currency symbol for the specified currency by using the currencySymbol attribute.

NOTE

You can obtain ISO4217, for a fee, from the ISO Web site (<http://www.iso.ch/>).

You can also find a Microsoft Word document containing these codes at

http://www.bsi-global.com/Technical+Information/Publications/_Publications/tig90.xalter. This site is the official ISO Web repository for the ISO4217 codes.

If the type attribute is percent or number, then you can use several number-formatting attributes. The maxIntegerDigits and minIntegerDigits attributes allow you to specify the size of the nonfractional portion of the number. If the actual number exceeds maxIntegerDigits, then the number is truncated.

Attributes are also provided to allow you to determine how many decimal places should be used. The minFractionalDigits and maxFractionalDigits attributes allow you to specify the number of decimal places. If the number exceeds the maximum number of fractional digits, the number will be rounded.

Grouping can be used to insert commas between thousands groups. Grouping is specified by setting the groupingIsUsed attribute to either true or false. When using grouping with minIntegerDigits, you must be careful to get your intended result. For example, if the value 1000 were instructed to have a minimum of 8 digits and grouping is used, the result would be

00,0001,000

If the previously mentioned formatting attributes are insufficient, you may elect to use the pattern attribute. This attribute lets you include special characters that specify how you would like your number encoded. [Table 6.1](#) shows these codes.

Table 6.1. Numeric Formatting Codes

Symbol	Meaning
--------	---------

0	Represents a digit.
#	Represents a digit; displays 0 as absent.
.	Serves as a placeholder for a decimal separator.
,	Serves as a placeholder for a grouping separator.
Separates formats.	
-	Used as the default negative prefix.
%	Multiplies by 100 and displays as a percentage.
?	Multiplies by 1000 and displays as per mille.
¤	Represents the currency sign; replaced by actional currency symbol.
X	Indicates that any other characters can be used in the prefix or suffix.
'	Used to quote special characters in a prefix or suffix.

Now that you have seen how to use the `<fmt:formatNumber>` tag, let's look at an example that uses this tag. Listing 6.1 shows a simple program that will format numbers. The program allows you to enter numbers and see them formatted in a variety of ways by JSTL. Figure 6.1 shows the output of our program.

Listing 6.1 Formatting Numbers (string.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt" %>
<html>
    <head>
        <title>Format Number</title>
    </head>

    <body>
        <form method="POST">
            <table border="1" cellpadding="0" cellspacing="0"
                style="border-collapse: collapse" bordercolor="#111111"
                width="62%" id="AutoNumber1">
                <tr>
                    <td width="100%" colspan="2" bgcolor="#0000FF">
                        <p align="center">
                            <b>
                                <font color="#FFFFFF" size="4">Number
                                Formatting</font>
                            </b>
                        </p>
                    </td>
                </tr>

                <tr>
                    <td width="47%">Enter a number to be formatted:</td>
                    <td width="53%">
                        <input type="text" name="num" size="20" />
                    </td>
                </tr>
            </table>
        </form>
    </body>
</html>
```

```

<tr>
    <td width="100%" colspan="2">
        <p align="center">
            <input type="submit" value="Submit" name="submit" />
            <input type="reset" value="Reset" name="reset" />
        </p>
    </td>
</tr>
</table>

<p>&#160;</p>
</form>

<c:if test="${pageContext.request.method=='POST' }">
    <table border="1" cellpadding="0" cellspacing="0"
style="border-collapse: collapse" bordercolor="#111111"
width="63%" id="AutoNumber2">
        <tr>
            <td width="100%" colspan="2" bgcolor="#0000FF">
                <p align="center">
                    <b>
                        <font color="#FFFFFF" size="4">Formatting:<br/>
                        <c:out value="${param.num}" />
                    </font>
                    </b>
                </p>
            </td>
        </tr>

        <tr>
            <td width="51%">type="number"</td>

            <td width="49%">
                <fmt:formatNumber type="number" value="${param.num}" />
            </td>
        </tr>

        <tr>
            <td>type="number" maxIntegerDigits="3"</td>

            <td>
                <fmt:formatNumber type="number" maxIntegerDigits="3"
value="${param.num}" />
            </td>
        </tr>

        <tr>
            <td>type="number" minIntegerDigits="10"</td>

            <td>
                <fmt:formatNumber type="number" minIntegerDigits="10"
value="${param.num}" />
            </td>
        </tr>

        <tr>

```

```

<td>type="number" maxFractionDigits="3"</td>

<td>
<fmt:formatNumber type="number" maxFractionDigits="3"
value="${param.num}" />
</td>
</tr>

<tr>
<td>type="number" minFractionDigits="10"</td>

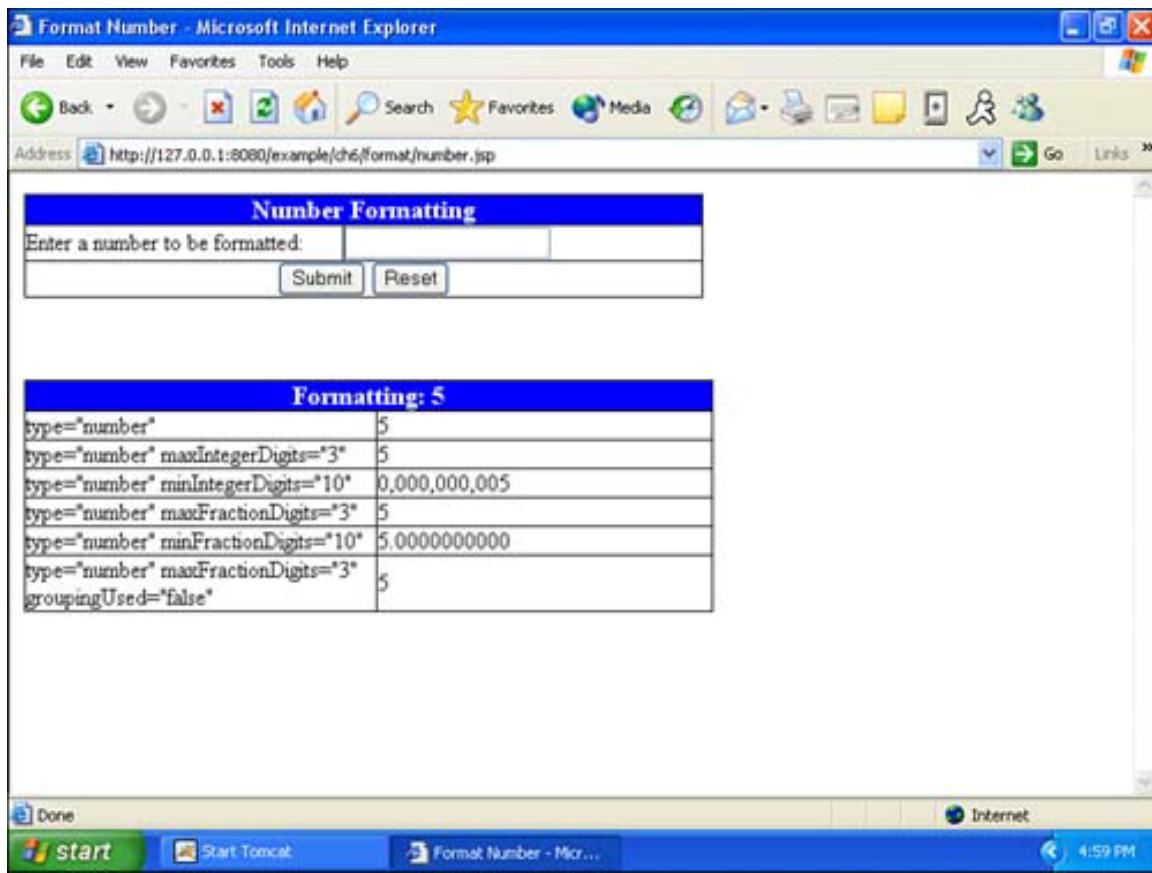
<td>
<fmt:formatNumber type="number" minFractionDigits="10"
value="${param.num}" />
</td>
</tr>

<tr>
<td>type="number" maxFractionDigits="3"
groupingUsed="false"</td>

<td>
<fmt:formatNumber type="number" maxFractionDigits="3"
groupingUsed="false" value="${param.num}" />
</td>
</tr>
</table>
</c:if>
</body>
</html>

```

Figure 6.1. Formatting numbers.



This program displays the number the user entered using variations of this

`<fmt:formatNumber>` tag:

```
<fmt:formatNumber type="number" maxFractionDigits="3"
    groupingUsed="false" value="${param.num}" />
```

It is possible to throw an exception using the `<fmt:formatNumber>` tag. If you enter a text value such as one, the program will throw an exception. If you would like to prevent such exceptions from stopping your program, you must use the `<c:catch>` tag that we discussed in [Chapter 4](#), "Using the Expression Language."

When you use very large numbers, rounding will sometimes occur, even though you have `maxFractionDigits` set to a large enough value to not require rounding. This odd behavior seems to occur only with very large numbers. Additionally, there appears to be an undocumented default of 3 for `maxFractionDigits`.

Formatting Percents

You can also format percents by using `<fmt:formatNumber>`. To format a percent, you must set the type attribute to percent. Numbers that are to be formatted as percent must be expressed in the conventional ratio form (for example, 50% is represented as .5).

Every other numeric formatting option that we described in the previous section also works with percent numbers. Listing 6.2 shows percent formatting applied to the same numeric formats used in our earlier example. Figure 6.2 shows the output from this program. As you can see, the percent symbol (%) is added. Whatever percentage indicator specified for the current locale will be used.

Listing 6.2 Formatting Percents (percent.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt" %>
<html>
    <head>
        <title>Format Percent</title>
    </head>

    <body>
        <form method="POST">
            <table border="1" cellpadding="0" cellspacing="0"
                style="border-collapse: collapse" bordercolor="#111111"
                width="62%" id="AutoNumber1">
                <tr>
                    <td width="100%" colspan="2" bgcolor="#0000FF">
                        <p align="center">
                            <b>
                                <font color="#FFFFFF" size="4">Number
                                    Formatting</font>
                            </b>
                        </p>
                    </td>
                </tr>

                <tr>
                    <td width="47%">Enter a percent to be formatted:</td>
                    <td width="53%">
                        <input type="text" name="num" size="20" />
                    </td>
                </tr>

                <tr>
                    <td width="100%" colspan="2">
                        <p align="center">
                            <input type="submit" value="Submit" name="submit" />
                            <input type="reset" value="Reset" name="reset" />
                        </p>
                    </td>
                </tr>
            </table>
            <p>&#160;</p>
        </form>

        <c:if test="${pageContext.request.method=='POST'}">
            <table border="1" cellpadding="0" cellspacing="0"
                style="border-collapse: collapse" bordercolor="#111111"
```

```

width="63%" id="AutoNumber2">
<tr>
  <td width="100%" colspan="2" bgcolor="#0000FF">
    <p align="center">
      <b>
        <font color="#FFFFFF" size="4">Formatting:</font>
        <c:out value="${param.num}" />
      </b>
    </p>
  </td>
</tr>

<tr>
  <td width="51%">type="percent"</td>

  <td width="49%">
    <fmt:formatNumber type="percent"
      value="${param.num}" />
  </td>
</tr>

<tr>
  <td>type="percent" maxIntegerDigits="3"</td>

  <td>
    <fmt:formatNumber type="percent" maxIntegerDigits="3"
      value="${param.num}" />
  </td>
</tr>

<tr>
  <td>type="percent" minIntegerDigits="10"</td>

  <td>
    <fmt:formatNumber type="percent" minIntegerDigits="10"
      value="${param.num}" />
  </td>
</tr>

<tr>
  <td>type="percent" maxFractionDigits="3"</td>

  <td>
    <fmt:formatNumber type="percent" maxFractionDigits="3"
      value="${param.num}" />
  </td>
</tr>

<tr>
  <td>type="percent" minFractionDigits="10"</td>

  <td>
    <fmt:formatNumber type="percent" minFractionDigits="10"
      value="${param.num}" />
  </td>
</tr>

```

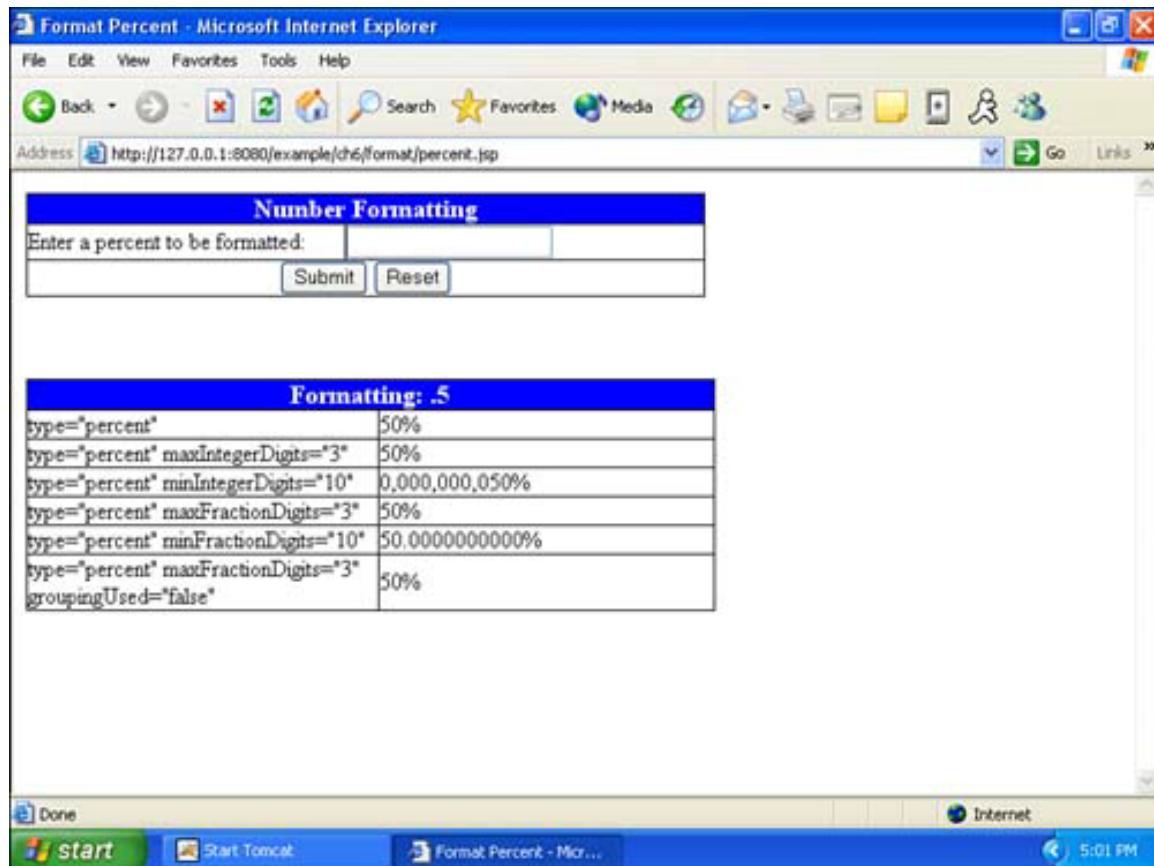
```

<tr>
    <td>type="percent" maxFractionDigits="3"
groupingUsed="false"</td>

    <td>
        <fmt:formatNumber type="percent" maxFractionDigits="3"
groupingUsed="false" value="${param.num}" />
    </td>
</tr>
</table>
</c:if>
</body>
</html>

```

Figure 6.2. Formatting percents.



Using the <fmt:formatDate> Tag

There are many occasions when a program must display dates. There are also many formats that a date can be displayed in. By using the `<fmt:formatDate>` tag, you can express your program's dates in a variety of ways. The `<fmt:formatDate>` tag is shown here:

```
<fmt:formatDate [value="date"]
    [type="{time|date|both}"]
    [dateStyle="{default|short|medium|long|full}"]
    [timeStyle="{default|short|medium|long|full}"]
    [pattern="customPattern"]
    [timeZone="timeZone"]
    [var="varName"]
    [scope="{page|request|session|application}"]/>
```

The attributes that are accepted by the `<fmt:formatDate>` tag include the following:

Attribute	Required	Purpose
<code>dateStyle</code>	N	The predefined formatting style for dates. Applied only when formatting a date or both a date and time (that is, if type is missing or is equal to date or both); ignored otherwise. This attribute uses DateFormat semantics.
<code>pattern</code>	N	The custom formatting style for dates and times.
<code>scope</code>	N	The scope of var.
<code>timeStyle</code>	N	The predefined formatting style for times. Applied only when formatting a time or both a date and time (that is, if type is missing or is equal to date or both); ignored otherwise.
<code>timeZone</code>	N	The time zone in which you want to represent the formatted time.
<code>type</code>	N	Specifies whether only the time, the date, or both the time and date components of the given date are to be formatted. This attribute defaults to date.
<code>value</code>	Y	The date and/or time to be formatted.
<code>var</code>	N	The name of the exported scoped attribute that stores the formatted result as a string. This attribute defaults to page.

In the first syntax, the value that is to be formatted is passed in using the value attribute. The formatted date will be written to the page if a var attribute is not specified. If a var attribute is specified, it will receive the formatted date as a string.

The second syntax uses the body of the tag to specify the date that is to be formatted. You should not provide a value attribute if you are using the tag with a body (syntax 2 in our example). In addition to these standard JSTL attributes, several formatting attributes are available specific to date formatting.

The type attribute allows you to specify whether this is a time, date, or both. The valid values for the type attribute are time, date, or both; date is the default. Additionally, you can specify the time and date styles using the timeStyle and dateStyle attributes. Both of these accept the values default, short, medium, long, or full. These values specify the amount of detail that will be displayed.

It is also possible to use the pattern attribute to specify even more precise handling of the date formatting. [Table 6.2](#) lists the format characters that can be used.

Table 6.2. Date-Formatting Codes

Code	Purpose	Sample Output
G	The era designator	AD
Y	The year	2002
M	The month	April & 04
d	The day of the month	20
h	The hour(12-hour time)	12
H	The hour(24-hour time)	0
m	The minute	45
s	The second	52
S	The millisecond	970
E	The day of the week	Tuesday
D	The day of the year	180
F	The day of the week in the month	2 (2nd Wed in month)
w	The week in the year	27
W	The week in the month	2
a	The a.m./p.m. indicator	PM
k	The hour(12-hour time)	24
K	The hour(24-hour time)	0
z	The time zone	Central Standard Time
'		The escape for text
''		The single quote

Using these codes gives you precise control of the output. For example, using the pattern string of hh 'o'clock' a, zzzz, you might get back 2 o'clock PM, Central Daylight Time. To get an idea of what some of these formats look like, let's examine Listing 6.3. The output from this program can be seen in Figure 6.3.

Listing 6.3 Formatting Dates (date.jsp)

```

<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jstl/core-rt" prefix="c-rt" %>
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt" %>
<html>
  <head>
    <title>Format Date</title>
  </head>

  <body>
    <c-rt:set var="now" value="<%=new java.util.Date()%>" />

    <table border="1" cellpadding="0" cellspacing="0"
      style="border-collapse: collapse" bordercolor="#111111"
      width="63%" id="AutoNumber2">

```

```

<tr>
  <td width="100%" colspan="2" bgcolor="#0000FF">
    <p align="center">
      <b>
        <font color="#FFFFFF" size="4">Formatting:</font>
        <fmt:formatDate value="${now}" type="both"
          timeStyle="long" dateStyle="long" />
      </b>
    </p>
  </td>
</tr>

<tr>
  <td width="51%">formatDate type="time"</td>

  <td width="49%">
    <fmt:formatDate type="time" value="${now}" />
  </td>
</tr>

<tr>
  <td width="51%">type="date"</td>

  <td width="49%">
    <fmt:formatDate type="date" value="${now}" />
  </td>
</tr>

<tr>
  <td width="51%">type="both"</td>

  <td width="49%">
    <fmt:formatDate type="both" value="${now}" />
  </td>
</tr>

<tr>
  <td width="51%">type="both" dateStyle="default"
    timeStyle="default"</td>

  <td width="49%">
    <fmt:formatDate type="both" dateStyle="default"
      timeStyle="default" value="${now}" />
  </td>
</tr>

<tr>
  <td width="51%">type="both" dateStyle="short"
    timeStyle="short"</td>

  <td width="49%">
    <fmt:formatDate type="both" dateStyle="short"
      timeStyle="short" value="${now}" />
  </td>
</tr>

```

```

<tr>
  <td width="51%">type="both" dateStyle="medium"
    timeStyle="medium"</td>

  <td width="49%">
    <fmt:formatDate type="both" dateStyle="medium"
      timeStyle="medium" value="${now}" />
  </td>
</tr>

<tr>
  <td width="51%">type="both" dateStyle="long"
    timeStyle="long"</td>

  <td width="49%">
    <fmt:formatDate type="both" dateStyle="long"
      timeStyle="long" value="${now}" />
  </td>
</tr>

<tr>
  <td width="51%">type="both" dateStyle="full"
    timeStyle="full"</td>

  <td width="49%">
    <fmt:formatDate type="both" dateStyle="full"
      timeStyle="full" value="${now}" />
  </td>
</tr>

<tr>
  <td width="51%">pattern="yyyy-MM-dd"</td>

  <td width="49%">
    <fmt:formatDate pattern="yyyy-MM-dd" value="${now}" />
  </td>
</tr>
</table>
</body>
</html>

```

Figure 6.3. Formatting dates.

The screenshot shows a Microsoft Internet Explorer window with the title bar "Format Date - Microsoft Internet Explorer". The address bar contains the URL "http://127.0.0.1:8080/example/ch6/format/date.jsp". The main content area displays a table titled "Formatting: April 28, 2002 5:01:51 PM CDT". The table lists various JSTL `<fmt:formatDate>` tag configurations and their resulting output. The columns are "formatDate type='time'" and "type='date'".

Formatting: April 28, 2002 5:01:51 PM CDT	
formatDate type="time"	5:01:51 PM
type="date"	Apr 28, 2002
type="both"	Apr 28, 2002 5:01:51 PM
type="both" dateStyle="default"	Apr 28, 2002 5:01:51 PM
timeStyle="default"	
type="both" dateStyle="short"	4/28/02 5:01 PM
timeStyle="short"	
type="both" dateStyle="medium"	Apr 28, 2002 5:01:51 PM
timeStyle="medium"	
type="both" dateStyle="long"	April 28, 2002 5:01:51 PM CDT
timeStyle="long"	
type="both" dateStyle="full"	Sunday, April 28, 2002 5:01:51 PM
timeStyle="full"	CDT
pattern="yyyy-MM-dd"	2002-04-28

The program in Listing 6.3 begins by obtaining the current date. This can be done with the following RT tag:

```
<c_rt:set var="now" value="<%=new java.util.Date()%>" />
```

This tag constructs a new Date object, which will contain the current time. This date is then assigned to the scoped variable now. Now that the date has been obtained, the program can display the date with a variety of `<fmt:formatDate>` tags.

Parsing

In addition to the formatting tags provided by JSTL, a second set of tags is available for parsing. In JSTL, parsing refers to tags that are used to access data entered by a user. As a result, it is quite possible for errors to occur if the data that is to be parsed is not in the format that the JSTL tag expects. In this section, we show you how to use the JSTL parsing tags.

Using the <fmt:parseNumber> Tag

The <fmt:parseNumber> tag is used to parse numbers, percentages, and currencies. There are two forms of this tag:

```
// Syntax 1: Without a body
<fmt:parseNumber value="numericValue"
  [type="{number|currency|percent}"]
  [pattern="customPattern"]
  [parseLocale="parseLocale"]
  [integerOnly="{true|false}"]
  [var="varName"]
  [scope="{page|request|session|application}"]/>
// Syntax 2: With a body to specify the numeric value to be parsed
<fmt:parseNumber [type="{number|currency|percent}"]
  [pattern="customPattern"]
  [parseLocale="parseLocale"]
  [integerOnly="{true|false}"]
  [var="varName"]
  [scope="{page|request|session|application}"]>
  numeric value to be parsed
</fmt:parseNumber>
```

The attributes accepted by the <fmt:parseNumber> tag include the following:

Attribute	Required	Purpose
integerOnly	N	Specifies whether just the integer portion of the given value should be parsed. This attribute defaults to true.
parseLocale	N	Specifies the locale whose default formatting pattern (for numbers, currencies, or percentages) is to be used during the parse operation, or to which the pattern specified via the pattern attribute (if present) is applied.
pattern	N	The custom formatting pattern that determines how the string in the value attribute is to be parsed. Applied only when parsing numbers (that is, if type is missing or is equal to number); ignored otherwise.
scope	N	The scope of var. This attribute defaults to page.
type	N	Specifies whether the string in the value attribute should be parsed as a number, currency, or percentage. This attribute defaults to number.
value	N	The string to be parsed.
var	N	The name of the exported scoped attribute that stores the parsed result (of type <code>java.lang.Number</code>).

In the first syntax, the value that is to be parsed is passed in using the value attribute. Once parsed, the number will be written to the page if a var attribute is not specified. If a var attribute is specified, it will receive the parsed number.

The second syntax uses the body of the tag to specify the number that is to be parsed. You should not provide a value attribute if you are using the tag with a body (syntax 2 in our example). In addition to these standard JSTL attributes, several parse attributes are available specific to number formatting.

A type attribute is provided to specify what type of number is to be parsed. Valid values for the type attribute are number, currency, and percent. The default is number.

A pattern attribute is provided that works just like the pattern attribute for the `<fmt:formatNumber>` tag. However, in the case of parsing, the pattern attribute tells the parser what format to expect. [Table 6.1](#) (earlier in the chapter) contains a complete list of pattern codes. The integerOnly attribute allows you to request that decimal places should not be parsed. Decimal places will not cause an error; they will simply be truncated.

The program in [Listing 6.4](#) shows some examples of formatting for different types.

Listing 6.4 Parsing Numbers, Currency, and Percents (parseNumber.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt" %>
<html>
    <head>
        <title>Parse Number</title>
    </head>

    <body>
        <form method="POST">
            <table border="1" cellpadding="0" cellspacing="0"
                style="border-collapse: collapse" bordercolor="#111111"
                width="62%" id="AutoNumber1">
                <tr>
                    <td width="100%" colspan="2" bgcolor="#0000FF">
                        <p align="center">
                            <b>
                                <font color="#FFFFFF" size="4">Number
                                    Formatting</font>
                            </b>
                        </p>
                    </td>
                </tr>

                <tr>
                    <td width="47%">Enter a number to be parsed:</td>
                    <td width="53%">
                        <input type="text" name="num" size="20" />
                    </td>
                </tr>

                <tr>
                    <td width="100%" colspan="2">
                        <p align="center">
                            <input type="submit" value="Submit" name="submit" />
                        </p>
                    </td>
                </tr>
            </table>
        </form>
    </body>
</html>
```

```

        <input type="reset" value="Reset" name="reset" />
    </p>
</td>
</tr>
</table>

<p>&#160;</p>
</form>

<c:if test="${pageContext.request.method=='POST'}">
    <table border="1" cellpadding="0" cellspacing="0"
    style="border-collapse: collapse" bordercolor="#111111"
    width="63%" id="AutoNumber2">
        <tr>
            <td width="100%" colspan="2" bgcolor="#0000FF">
                <p align="center">
                    <b>
                        <font color="#FFFFFF" size="4">Formatting:<br/>
                            <c:out value="${param.num}" />
                        </font>
                    </b>
                </p>
            </td>
        </tr>

        <tr>
            <td width="51%">type="number"</td>

            <td width="49%">
                <c:catch var="e">
                    <fmt:parseNumber var="i" type="number"
                    value="${param.num}" />

                    <c:out value="${i}" />
                </c:catch>

                <c:out value="${e}" escapeXml="false" />
            </td>
        </tr>

        <tr>
            <td width="51%">type="currency"</td>

            <td width="49%">
                <c:catch var="e">
                    <fmt:parseNumber var="i" type="currency"
                    value="${param.num}" />

                    <c:out value="${i}" />
                </c:catch>

                <c:out value="${e}" escapeXml="false" />
            </td>
        </tr>

        <tr>

```

```

<td width="51%">type="percent"</td>

<td width="49%">
<c:catch var="e">
<fmt:parseNumber var="i" type="percent"
value="\${param.num}" />

<c:out value="\${i}" />
</c:catch>

<c:out value="\${e}" escapeXml="false" />
</td>
</tr>

<tr>
<td width="51%">type="number" integerOnly="true"</td>

<td width="49%">
<c:catch var="e">
<fmt:parseNumber var="i" integerOnly="true"
type="number" value="\${param.num}" />

<c:out value="\${i}" />
</c:catch>

<c:out value="\${e}" escapeXml="false" />
</td>
</tr>
</table>
</c:if>
</body>
</html>

```

The program in Listing 6.4 works by asking the user to enter a number. This number will be then parsed in several ways. Because it is possible that the user might enter an invalid number, each call to the `<fmt:parseNumber>` tag is bounded in `<c:catch>` tags. The following code shows how this is done:

```

<c:catch var="e">
<fmt:parseNumber var="i" integerOnly="true"
type="number" value="\${param.num}" />
<c:out value="\${i}" />
</c:catch>

```

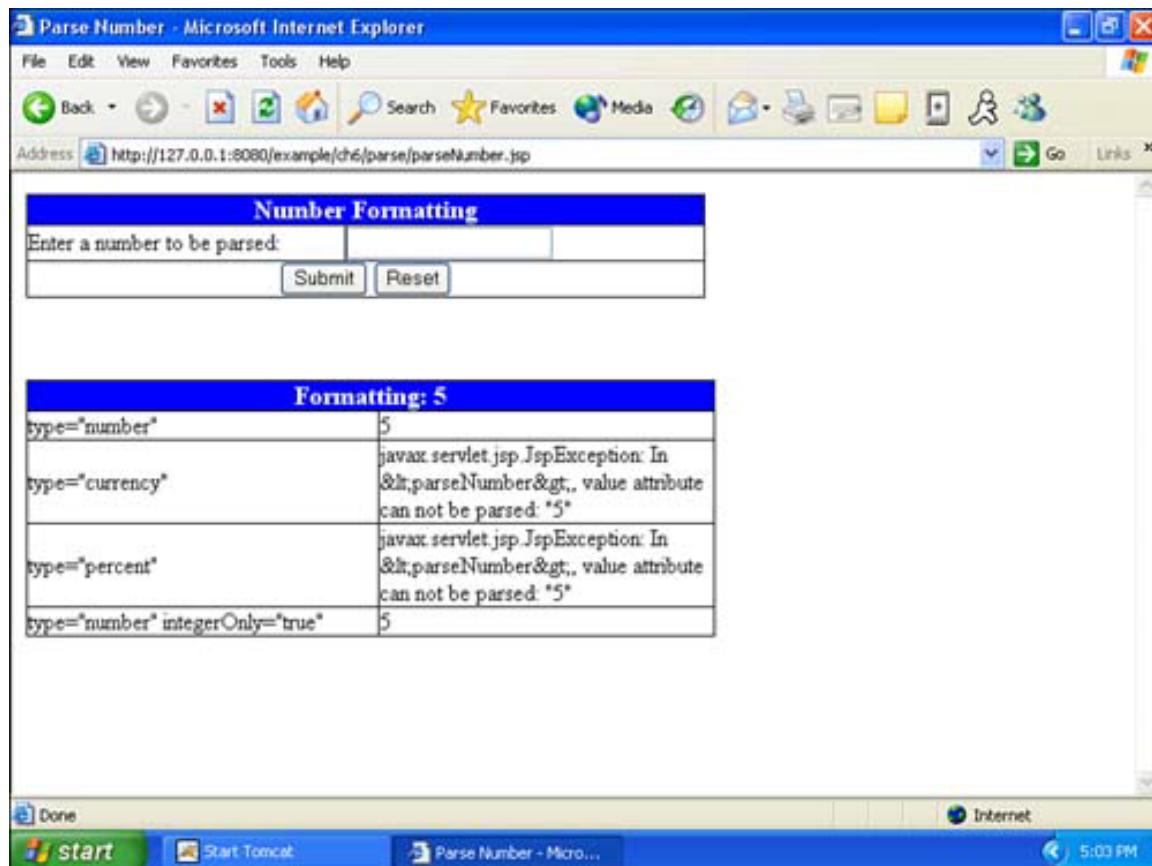
This code attempts to pass the scoped variable `i` as a number. If the scoped variable `i` cannot be parsed as a variable, an exception is thrown. This exception, if thrown, will be placed in the `e` scoped variable. If any exception is thrown, the following line of code prints it out:

```
<c:out value="\${e}" escapeXml="false" />
```

Our program also attempts to parse as a percent and currency. Usually, several of the parses in this program will throw an exception, since it is impossible for one number to

satisfy all types. The output from this program, on a typical number, is shown in Figure 6.4.

Figure 6.4. Parsing numbers, percents, and currencies.



Using the <fmt:parseDate> Tag

There are many occasions when a program must display dates. There are also many formats that a date can be displayed in. By using the `<fmt:parseDate>` tag, you can express your program's dates in a variety of ways. The two forms of the `<fmt:parseDate>` tag are as follows:

```
// Syntax 1: Without a body
<fmt:parseDate value="dateString"
    [type="{time|date|both}"]
    [dateStyle="{default|short|medium|long|full}"]
    [timeStyle="{default|short|medium|long|full}"]
    [pattern="customPattern"]
    [timeZone="timeZone"]
    [parseLocale="parseLocale"]
    [var="varName"]
    [scope="{page|request|session|application}"]/>
// Syntax 2: With a body to specify the date value to be parsed
<fmt:parseDate [type="{time|date|both}"]>
```

```
[dateStyle="{default|short|medium|long|full}"]
[timeStyle="{default|short|medium|long|full}"]
[pattern="customPattern"]
[timeZone="timeZone"]
[parseLocale="parseLocale"]
[var="varName"]
[scope="{page|request|session|application}"]>
    date value to be parsed
</fmt:parseDate>
```

The attributes that are accepted by the `<fmt:parseDate>` tag include the following:

Attribute	Required	Purpose
<code>dateStyle</code>	N	The style for the date. Should be default, short, long, medium, or full. The default is default. This attribute uses <code>DateFormat</code> semantics.
<code>parseLocale</code>	N	Specifies the locale that should be used. This attribute should be of type <code>String</code> or <code>java.util.Locale</code> .
<code>pattern</code>	N	Specifies how the date is to be parsed using a pattern.
<code>timeStyle</code>	N	The style for the time. Should be default, short, long, medium, or full. The default is default.
<code>timeZone</code>	N	The zone that the parsed time is from. This value should be of type <code>String</code> or <code>java.util.TimeZone</code> .
<code>type</code>	N	The type of date to parse. Should be time, date, or both. The default is date.
<code>value</code>	N	The string to be parsed.
<code>var</code>	N	The name of the exported scoped attribute that stores the parsed result (of type <code>java.lang.Number</code>).

In the first syntax, the value that is to be parsed is passed using the `value` attribute. The parsed date will be written to the page if a `var` attribute is not specified. If a `var` attribute is specified, it will receive the parsed date.

The second syntax uses the body of the tag to specify the date that is to be parsed. You should not provide a `value` attribute if you are using the tag with a body (syntax 2 in our example). In addition to these standard JSTL attributes, several formatting attributes are available specific to date parsing.

The `type` attribute allows you to specify whether a time, date, or both will be parsed. The valid values for the `type` attribute are `time`, `date`, and `both`. A `pattern` attribute is provided that works just like the `pattern` attribute for the `<fmt:formatDate>` tag. However, in the case of parsing, the `pattern` attribute tells the parser what format to expect. A complete list of pattern codes can be found in [Table 6.2](#). A `timeZone` attribute is available to specify the time zone; we cover time zones in the next section.

The `dateStyle` and `timeStyle` attributes are specified, just as with the `<fmt:formatDate>` tag. These tags indicate which format the date will be in. Refer to Figure 6.3 to see the various formats available. Listing 6.5 shows a program that demonstrates parsing dates and times.

Listing 6.5 Parsing Dates (parseDate.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt" %>
<html>
    <head>
        <title>Parse Date</title>
    </head>

    <body>
        <form method="POST">
            <table border="1" cellpadding="0" cellspacing="0"
                style="border-collapse: collapse" bordercolor="#111111"
                width="62%" id="AutoNumber1">
                <tr>
                    <td width="100%" colspan="2" bgcolor="#0000FF">
                        <p align="center">
                            <b>
                                <font color="#FFFFFF" size="4">Date
                                    Formatting</font>
                            </b>
                        </p>
                    </td>
                </tr>

                <tr>
                    <td width="47%">Enter a date to be parsed:</td>
                    <td width="53%">
                        <input type="text" name="num" size="20" />
                    </td>
                </tr>

                <tr>
                    <td width="100%" colspan="2">
                        <p align="center">
                            <input type="submit" value="Submit" name="submit" />
                            <br>
                            <input type="reset" value="Reset" name="reset" />
                        </p>
                    </td>
                </tr>
            </table>
        <p>&#160;</p>
    </form>

    <c:if test="${pageContext.request.method=='POST'}">
        <table border="1" cellpadding="0" cellspacing="0"
            style="border-collapse: collapse" bordercolor="#111111"
```

```

width="63%" id="AutoNumber2">
<tr>
  <td width="100%" colspan="2" bgcolor="#0000FF">
    <p align="center">
      <b>
        <font color="#FFFFFF" size="4">Formatting:</font>
        <c:out value="${param.num}" />
      </b>
    </p>
  </td>
</tr>

<tr>
  <td width="51%">type="date" dateStyle="short"</td>

  <td width="49%">
    <c:catch var="e">
      <fmt:parseDate var="i" type="date" dateStyle="short"
      value="${param.num}" />

      <c:out value="${i}" />
    </c:catch>

      <c:out value="${e}" escapeXml="false" />
  </td>
</tr>

<tr>
  <td width="51%">type="date" dateStyle="medium"</td>

  <td width="49%">
    <c:catch var="e">
      <fmt:parseDate var="i" type="date" dateStyle="medium"
      value="${param.num}" />

      <c:out value="${i}" />
    </c:catch>

      <c:out value="${e}" escapeXml="false" />
  </td>
</tr>

<tr>
  <td width="51%">type="date" dateStyle="long"</td>
  <td width="49%">
    <c:catch var="e">
      <fmt:parseDate var="i" type="date" dateStyle="long"
      value="${param.num}" />

      <c:out value="${i}" />
    </c:catch>

      <c:out value="${e}" escapeXml="false" />
  </td>
</tr>

```

```

<tr>
    <td width="51%">type="date" dateStyle="full"</td>

    <td width="49%">
        <c:catch var="e">
            <fmt:parseDate var="i" type="date" dateStyle="full"
                           value="${param.num}" />

            <c:out value="${i}" />
        </c:catch>

        <c:out value="${e}" escapeXml="false" />
    </td>
</tr>
</table>
</c:if>
</body>
</html>

```

The program in Listing 6.5 works by asking the user to enter a date. This date will be then parsed in several ways. Because it is possible that the user might enter an invalid date, each call to the `<fmt:parseDate>` tag is bounded in `<c:catch>` tags. The following code shows how this is done:

```

<c:catch var="e">
    <fmt:parseDate var="i" type="date" dateStyle="full"
                   value="${param.num}" />
    <c:out value="${i}" />
</c:catch>

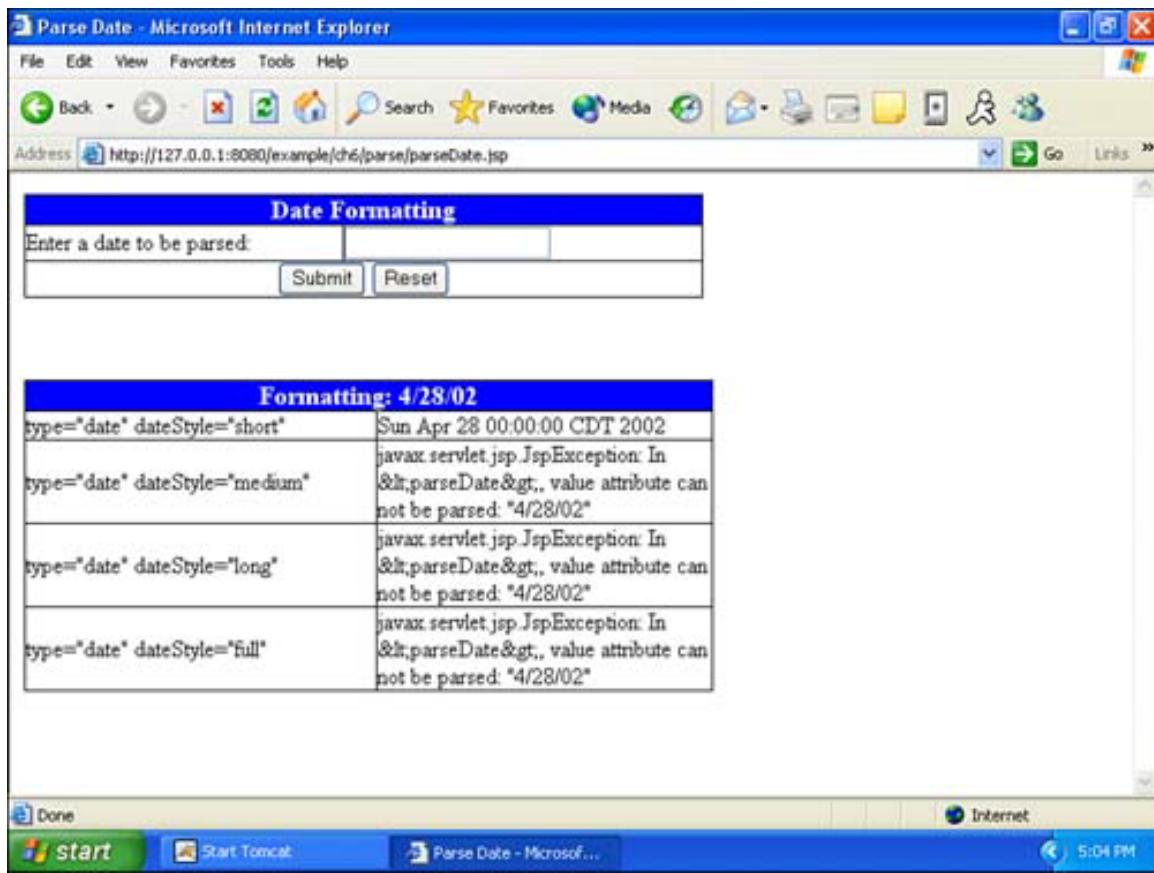
```

This code attempts to pass the scoped variable `i` as a date. If the scoped variable `i` cannot be parsed as a date, then an exception is thrown. This exception, if thrown, will be placed in the `e` scoped variable. The following line of code prints it out:

```
<c:out value="${e}" escapeXml="false" />
```

This program also attempts to parse several formats. Usually, several of the parses in the program will throw an exception, since it is impossible for one date to satisfy all types. The output from this program, on a typical number, is shown in Figure 6.5.

Figure 6.5. Parsing dates.



Time Zones

Nearly every Web application must be concerned with time zones. It is very likely that the time zone that your Web server is in is different from the time zone of the user accessing your Web site. JSTL gives you several tags that allow you to set the time zones that date formatting tags deal with.

Setting Time Zones

There are two ways that you can set the time zone in a JSTL-based application. The first is by modifying the web.xml file for the Web site so that it specifies a default time zone. The second is to use the `<fmt:TimeZone>` tag.

Using the `<fmt:TimeZone>` Tag

It is also possible to set the time zone for a specific range of tags. The `<fmt:TimeZone>` will specify the time zone that all tags within its body will use. This can allow you to customize the time zone on a per-user basis. Here is the format of the `<fmt:TimeZone>` tag:

```
<fmt:timeZone value="timeZone"
 [var="varName"]
body content
</fmt:timeZone>
```

The `<fmt:timeZone>` tag accepts the following attribute:

Attribute	Required	Purpose
value	Y	Specifies the time zone that should be used by the tags enclosed in the body of this tag.

The body content of the `<fmt:timeZone>` tag specifies the tags that will be controlled by this time zone. For example, the following tags would use the U.S. Central time zone:

```
<fmt:timeZone value="CDT">
    <t:formatDate value="${now}" type="both" />
</fmt:timeZone>
```

We will show you a program later in this chapter that makes use of multiple time zones.

Specifying the Time Zone in web.xml

If you take no actions to influence the time zone, JSTL will use the time zone that your computer itself is in. If you would like to override your machine's time zone, you can use the `web.xml` file. The following lines from a `web.xml` file attempt to set the default time zone to Central time:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
<context-param>
    <param-name>javax.servlet.jsp.jstl.fmt.timeZone</param-name>
    <param-value>US/Central</param-value>
</context-param>
```

Using the <fmt:setTimeZone> Tag

It is possible to load a time zone into an individual scoped variable. This variable will then be passed to the `<fmt:timeZone>` tag. The `<fmt:setTimeZone>` tag is used to copy a time zone object into the specified scoped variable. Here is the format of the `<fmt:setTimeZone>` tag:

```
<fmt:setTimeZone value="timeZone"
 [var="varName"]
 [scope="{page|request|session|application}"] />
```

The attributes accepted by the `<fmt:setTimeZone>` tag include the following:

Attribute	Required	Purpose
scope	N	Specifies the scope of scoped variable specified by var. Defaults to page.
value	Y	Specifies the time zone that should be used by the tags enclosed in the body of this tag.
var	N	Specifies the scoped variable that is to receive the time zone. If this is not given, a new default time zone will be set.

As you can see, you must specify the time zone using the value attribute. This is done in the same way the time zone is specified using the `<fmt:timeZone>` tag. For example, specifying the attribute CDT would select the time zone for Central U.S. time. This time zone will then be stored in the variable specified by the var scoped variable. Let's look at a program that makes use of multiple time zones.

Using Time Zones

[Listing 6.6](#) shows a program that will display all of the time zones that Java has access to. The current time will be translated into each of these time zones.

Listing 6.6 Displaying Time Zones (zone.jsp)

```

<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jstl/core-rt" prefix="c-rt" %>
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt" %>
<html>
    <head>
        <title>Timezones</title>
    </head>

    <body>
        <c-rt:set var="now" value="<%=new java.util.Date()%>" />

        <table border="1" cellpadding="0" cellspacing="0"
            style="border-collapse: collapse" bordercolor="#111111"
            width="63%" id="AutoNumber2">
            <tr>
                <td width="100%" colspan="2" bgcolor="#0000FF">
                    <p align="center">
                        <b>
                            <font color="#FFFFFF" size="4">Formatting:<br>
                            <fmt:formatDate value="${now}" type="both">
                                timeStyle="long" dateStyle="long" />
                            </font>
                        </b>
                    </p>
                </td>
            </tr>

            <c-rt:forEach var="zone"
                items="<%=java.util.TimeZone.getAvailableIDs()%>">
                <tr>

```

```

<td width="51%">
    <c:out value="${zone}" />
</td>

<td width="49%">
    <fmt:timeZone value="${zone}">
        <fmt:formatDate value="${now}" timeZone="${zn}" type="both" />
    </fmt:timeZone>
</td>
</tr>
</c-rt:forEach>
</table>
</body>
</html>

```

The program in Listing 6.6 goes through every time zone that Java has available. This is done by calling the method `java.util.TimeZone.getAvailableIDs()`. The following code begins this process:

```
<c-rt:forEach var="zone"
    items="<%java.util.TimeZone.getAvailableIDs()%>">
```

For each pass through this program, the time zone is displayed:

```
<c:out value="${zone}" />
```

Finally, the current time is converted to the new time zone. The following code does this by using the `<fmt:timeZone>` tag:

```
<fmt:timeZone value="${zone}">
    <fmt:formatDate value="${now}" timeZone="${zn}" type="both" />
</fmt:timeZone>
</c-rt:forEach>
```

Our program displays a lengthy list of time zones. You can see the program's output in Figure 6.6.

Figure 6.6. Displaying time zones.

The screenshot shows a Microsoft Internet Explorer window with the title "Timezones - Microsoft Internet Explorer". The address bar contains the URL "http://127.0.0.1:8080/example/ch6/date/zone.jsp". The main content is a table titled "Formatting: April 28, 2002 5:05:36 PM CDT". The table lists various time zones and their corresponding dates and times. The table has 2 columns and 26 rows. The first column contains the time zone names and the second column contains the dates and times.

Formatting: April 28, 2002 5:05:36 PM CDT	
Etc/GMT+12	Apr 28, 2002 10:05:36 AM
Etc/GMT+11	Apr 28, 2002 11:05:36 AM
MIT	Apr 28, 2002 11:05:36 AM
Pacific/Apia	Apr 28, 2002 11:05:36 AM
Pacific/Midway	Apr 28, 2002 11:05:36 AM
Pacific/Nuue	Apr 28, 2002 11:05:36 AM
Pacific/Pago_Pago	Apr 28, 2002 11:05:36 AM
Pacific/Samoa	Apr 28, 2002 11:05:36 AM
US/Samoa	Apr 28, 2002 11:05:36 AM
America/Adak	Apr 28, 2002 1:05:36 PM
America/Atka	Apr 28, 2002 1:05:36 PM
Etc/GMT+10	Apr 28, 2002 12:05:36 PM
HST	Apr 28, 2002 12:05:36 PM
Pacific/Fakaofo	Apr 28, 2002 12:05:36 PM
Pacific/Honolulu	Apr 28, 2002 12:05:36 PM
Pacific/Johnston	Apr 28, 2002 12:05:36 PM
Pacific/Rarotonga	Apr 28, 2002 12:05:36 PM
Pacific/Tahiti	Apr 28, 2002 12:05:36 PM
SystemV/HST10	Apr 28, 2002 12:05:36 PM
ITS/Alentian	Apr 28, 2002 1:05:36 PM

It is important to make sure that you display the correct time zone whenever possible. If your Web application has anonymous users, then the best you can do is use the local time zone of the server. However, if your system maintains user accounts, you may wish to allow the user to specify a default time zone. In this way, your program could display all times in the time zone for that user. Keep in mind that this is only the time zone that the user uses, though. All internal times stored to your database, or other persistent storage, should be stored in a common time zone. This time zone will usually either be the current time zone of your server or GMT.

Applying Date Formatting

Now that you know how to format dates and times in various ways, let's see how to apply these tags. Listing 6.7 shows a program that displays a calendar.

Listing 6.7 Displaying a Calendar (calendar.jsp)

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

```

<title>Calendar</title>
</head>

<body>
    <form method="POST">
        <table border="1" cellpadding="0" cellspacing="0"
            style="border-collapse: collapse" bordercolor="#111111"
            width="62%" id="AutoNumber1">
            <tr>
                <td width="100%" colspan="2" bgcolor="#0000FF">
                    <p align="center">
                        <b>
                            <font color="#FFFFFF" size="4">Date
                                Formatting</font>
                        </b>
                    </p>
                </td>
            </tr>

            <tr>
                <td width="47%">Enter a month(1-12)</td>
                <td width="53%">
                    <input type="text" name="month" size="20" />
                </td>
            </tr>

            <tr>
                <td width="47%">Enter a year(i.e. 2002)</td>
                <td width="53%">
                    <input type="text" name="year" size="20" />
                </td>
            </tr>

            <tr>
                <td width="100%" colspan="2">
                    <p align="center">
                        <input type="submit" value="Submit" name="submit" />
                        <input type="reset" value="Reset" name="reset" />
                    </p>
                </td>
            </tr>
        </table>
        <p>&#160;</p>
    </form>

    <c:if test="${pageContext.request.method=='POST' }">
        <table border="1" cellpadding="0" cellspacing="0"
            style="border-collapse: collapse" bordercolor="#111111"
            width="63%" id="AutoNumber2">
            <fmt:parseDate var="now"
                value="${param.month}/1/${param.year}" type="date"
                dateStyle="short" />

```

```

<tr>
  <td width="100%" colspan="7" bgcolor="#0000FF">
    <p align="center">
      <b>
        <font color="#FFFFFF" size="4">
          <fmt:formatDate pattern="MMMMM YYYY"
            value="${now}" />
        </font>
      </b>
    </p>
  </td>
</tr>

<fmt:formatDate var="i" pattern="E" value="${now}" />

<c:choose>
  <c:when test="${i=='Sun'}">
    <c:set var="i" value="1" />
  </c:when>

  <c:when test="${i=='Mon'}">
    <c:set var="i" value="2" />
  </c:when>

  <c:when test="${i=='Tue'}">
    <c:set var="i" value="3" />
  </c:when>

  <c:when test="${i=='Wed'}">
    <c:set var="i" value="4" />
  </c:when>

  <c:when test="${i=='Thu'}">
    <c:set var="i" value="5" />
  </c:when>

  <c:when test="${i=='Fri'}">
    <c:set var="i" value="6" />
  </c:when>

  <c:when test="${i=='Sat'}">
    <c:set var="i" value="7" />
  </c:when>

  <c:otherwise>
    <c:set var="i" value="?" />
  </c:otherwise>
</c:choose>

<c:choose>
  <c:when test="${param.month==2}">
    <c:set var="max" value="28" />

    <c:if
      test="${(param.year % 4 == 0 && param.year % 100 != 0)
      || param.year % 400 == 0 }">

```

```

        <c:set var="max" value="29" />
    </c:if>
</c:when>

<c:when test="${param.month==4}">
    <c:set var="max" value="30" />
</c:when>

<c:when test="${param.month==6}">
    <c:set var="max" value="30" />
</c:when>

<c:when test="${param.month==9}">
    <c:set var="max" value="30" />
</c:when>

<c:when test="${param.month==11}">
    <c:set var="max" value="30" />
</c:when>

<c:otherwise>
    <c:set var="max" value="31" />
</c:otherwise>
</c:choose>

<tr>
    <td width="70">
        <b>
            <center>Sunday</center>
        </b>
    </td>

    <td width="70">
        <b>
            <center>Monday</center>
        </b>
    </td>

    <td width="70">
        <b>
            <center>Tuesday</center>
        </b>
    </td>

    <td width="70">
        <b>
            <center>Wednesday</center>
        </b>
    </td>

    <td width="70">
        <b>
            <center>Thursday</center>
        </b>
    </td>

    <td width="70">

```

```

<b>
    <center>Friday</center>
</b>
</td>

<td width="70">
<b>
    <center>Saturday</center>
</b>
</td>
</tr>

<c:set var="d" value="1" />

<c:set var="d" value="1" />

<c:forEach var="x" begin="1" end="35">
    <c:if
        test="${(x==1) || (x==8) || (x==15) || (x==22) || (x==29) }">
        </tr><tr>
    </c:if>

    <td>
    <c:if
        test="${(d<=max) && ((x>7) || (i<=x)) }">
            <c:out value="${d}" />
            <c:set var="d" value="${d+1}" />
        </c:if>

        &#160;
        <br />
        <br />
        <br />
    </td>
    </c:if>
</c:forEach>
</table>
</c:if>
</body>
</html>

```

The program in Listing 6.7 displays a calendar page for any specified month and year. To do this, the program must first determine what day of the week (for example, Monday) the first day of the specified month falls on. The following code accomplishes this:

```

<fmt:parseDate var="now"
value="${param.month}/1/${param.year}" type="date"
dateStyle="short" />

```

This code constructs a new date that is the "first day" of the specified year and month. We can then request the day of the week from the new date, as this line demonstrates:

```
<fmt:formatDate var="i" pattern="E" value="${now}" />
```

By requesting the formatting character "E", we are given the day of the week (Mon, Tue, and so forth). We must now convert the text of the day of the week to a "column number." This tells us which column to start numbering on when the calendar is displayed. The following code does this conversion for the first two weekdays:

```
<c:when test="${i=='Sun'}">
  <c:set var="i" value="1" />
</c:when>
<c:when test="${i=='Mon'}">
  <c:set var="i" value="2" />
</c:when>
```

Now that we know what column number to start on, we know where to begin. Next, we must decide where to end, so we have to determine how many days are in the requested month. For most months, this is easy; there is a constant number of days for each month. The problem lies in February, which changes its length from 28 to 29 when the current year is a leap year. To handle February, we must determine whether this is a leap year.

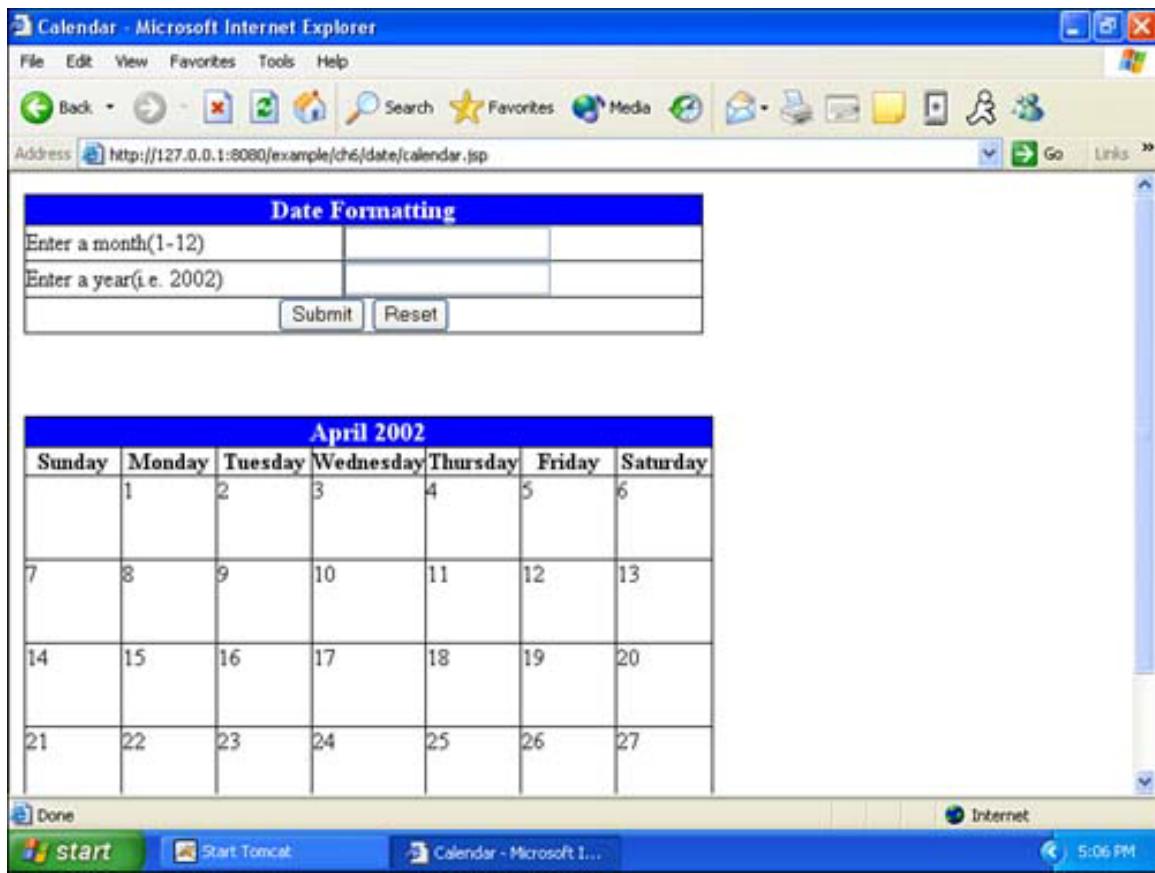
This algorithm is relatively simple. A year is a leap year if it is divisible by 4. But if the year is also divisible by 100, then it is not a leap year unless it is divisible by 400. Years such as 1992 and 1996 are leap years because they are divisible by 4 and are not affected by the rest of the rule. The remaining part of the rule applies to century years, such as 1900 and 2000. Century years are not leap years except when they are a multiple of 400. Because of this, the years 1700, 1800, and 1900 were not leap years. But the infamous year 2000 was a leap year. Translated to a JSTL expression, this becomes:

```
<c:choose>
  <c:when test="${param.month==2}">
    <c:set var="max" value="28" />

    <c:if
      test="${(param.year % 4 == 0 && param.year % 100 != 0) ||
      param.year % 400 == 0 }">
      <c:set var="max" value="29" />
    </c:if>
  </c:when>
  <c:when test="${param.month==4}">
    <c:set var="max" value="30" />
  </c:when>
```

As you can see, an ordinary month such as April, month number four, is simply assigned 30 days. However, for February, month two, the leap year rule is applied. We have now determined the beginning and ending spots on the calendar grid for the requested calendar. The calendar is then drawn to an HTML table using a `<c:forEach>` tag. The output from the calendar program is shown in [Figure 6.7](#).

Figure 6.7. Displaying a calendar.



Summary

In this chapter, we saw how you can format and parse data by using JSTL tags. Up to this point, the only source of data that we have examined has been data entered by the user. In subsequent chapters, we examine data from other sources. In the next three chapters, we show you how to obtain data through XML, SQL, and the Internet.

Chapter 7. Accessing Data with SQL Tags

Data access is an important feature of any Web application. Most Web applications are driven by SQL-compatible databases. JSTL provides a complete set of tags that allows access to SQL databases.

You may be wondering why such tags are needed. Most multitier Web applications are designed to keep database access out of JSP pages completely. This allows JSP pages to concentrate on the presentation interface while a collection of Enterprise JavaBeans (EJB) handles the backend database access. The JSTL SQL tags are meant to give you a way to incorporate SQL access into smaller projects or prototypes that do not need the complexity introduced by a multitier system.

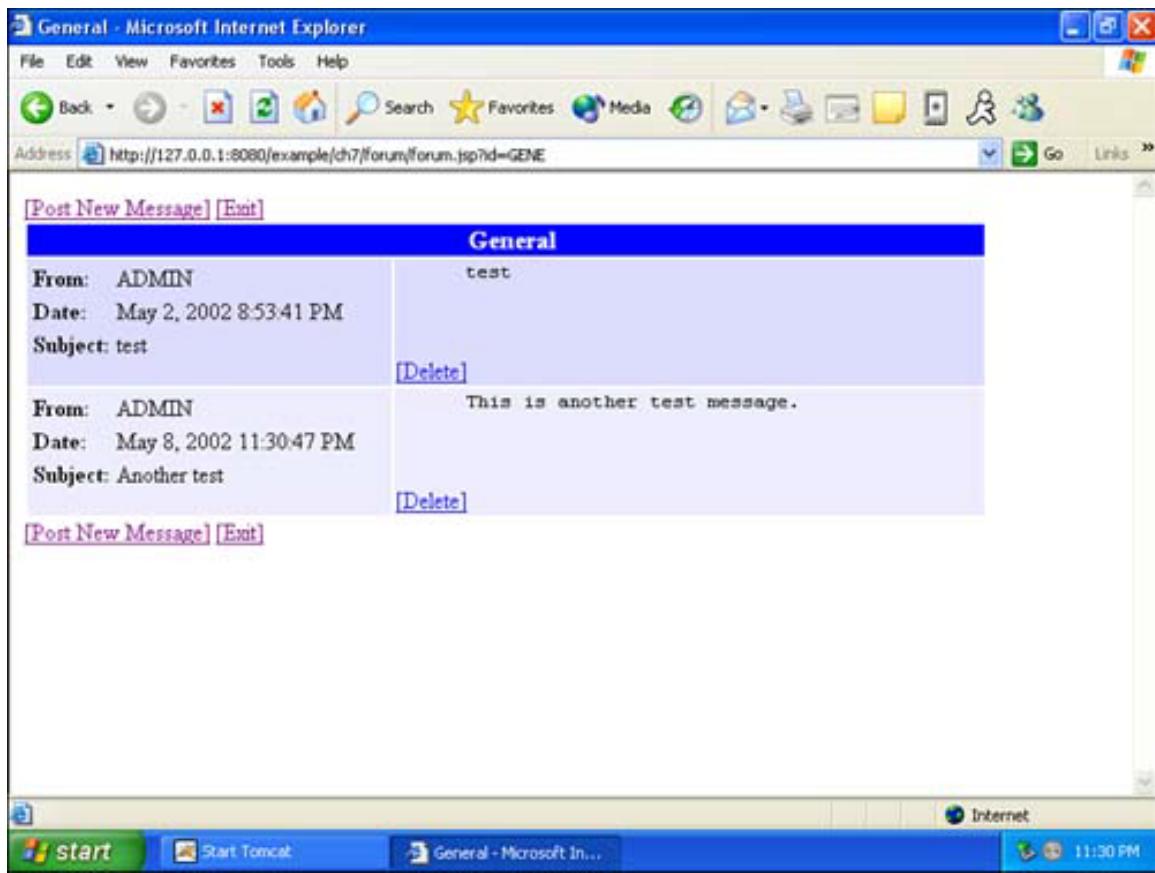
We begin by examining how to create a Web application using only JSTL tags. [Chapter 11](#), "Creating Your Own Tag Libraries," will follow up by showing you how to move this functionality into your own programs.

This chapter will present a complete Web application that was written entirely in JSTL. The primary examples for the next several chapters in this book will build on this Web application. By the end of [Chapter 12](#), you will have developed a multilingual JSTL database application that does all backend processing through a custom tag library. This is the model that Sun is promoting with JSP 1.3.

Introducing the Forum Example

Most Internet users are familiar with online forums. These systems go by many names—message boards, discussion threads, newsgroups, bulletin boards. Online forums allow users to post messages to an area so the messages can be viewed by other users. Users can post messages as well as responses to other people's messages. [Figure 7.1](#) shows our forum application.

Figure 7.1. Our forum application.



As you can see, this Web application allows users to post messages. To accomplish this, the application requires several interrelated services:

- User security
- New user registration
- Message posting
- Administration activities

Our forum application provides each of these services using JSTL SQL tags. One common thread to all of these services is the underlying database.

Database Layout

Behind every good Web application is a database structure. Our forum application is no exception. In this section, we discuss the structure of the forum database. This structure will remain the same throughout the multiple versions of this application that we develop in the next few chapters.

The forum application has been tested primarily with MySQL and Microsoft Access. You can obtain drivers for both at no charge. For more information on how to install MySQL, refer to [Appendix C](#), "Installing MySQL."

NOTE

Even if you do not own a copy of Microsoft Access, you can download the necessary drivers from Microsoft. With these drivers installed, you will be able to run the Access examples presented in this chapter. Download these drivers from <http://www.microsoft.com/data/download.htm>.

Setting Up the Forum on SQL Databases

First, we examine how to set up the forum on a SQL database. To properly set up your database for the forum, you must create a database named forum and assign a user named forumuser to that database. The procedure for creating databases in MySQL is covered in [Appendix C](#).

Once you've created this database, you must create the appropriate tables. You can do this using the database-creation script shown in [Listing 7.1](#). Executing the script against your database will set up the appropriate tables. We tested our script with MySQL; if you are using another database, you may have to make some minor changes.

Listing 7.1 Forum-Creation Script (createforum.sql)

```
DROP DATABASE IF EXISTS forum;
CREATE DATABASE forum;
USE forum;

CREATE TABLE t_access (
    c_access VARCHAR(1) NOT NULL,
    c_description VARCHAR(32) NOT NULL,
    PRIMARY KEY (c_access)
);

INSERT INTO t_access(c_access,c_description)
VALUES(
    'A', 'Admin User'
);

INSERT INTO t_access(c_access,c_description)
VALUES(
    'R', 'Regular User'
);

INSERT INTO t_access(c_access,c_description)
VALUES(
    'G', 'Guest User'
);

CREATE TABLE t_users(
    c_uid VARCHAR(32) NOT NULL,
    c_pwd VARCHAR(32) NOT NULL,
```

```

c_accesses INTEGER NOT NULL,
c_first DATETIME NOT NULL,
c_last DATETIME NOT NULL,
c_bad INTEGER NOT NULL,
c_posted INTEGER NOT NULL,
c_type VARCHAR(1) NOT NULL,
PRIMARY KEY(c_uid),
FOREIGN KEY(c_type) REFERENCES t_access(c_access)
);

INSERT INTO
t_users(c_uid,c_pwd,c_accesses,c_first,c_last,c_bad,c_posted,c_type)
values(
'ADMIN','admin',0,'04-01-02','2002-04-01',0,0,'A'
);

INSERT INTO
t_users(c_uid,c_pwd,c_accesses,c_first,c_last,c_bad,c_posted,c_type)
values(
'TESTUSER','test',0,'04-01-02','2002-04-01',0,0,'A'
);

INSERT INTO
t_users(c_uid,c_pwd,c_accesses,c_first,c_last,c_bad,c_posted,c_type)
values(
'TESTGUEST','test',0,'04-01-02','2002-01-02',0,0,'A'
);

CREATE TABLE t_forums(
c_code VARCHAR(4) NOT NULL,
c_name VARCHAR(50) NOT NULL,
c_sequence INTEGER NOT NULL,
PRIMARY KEY(c_code)
);

INSERT INTO t_forums(c_code,c_name,c_sequence)
values(
'GNRL','General Discussion',1
);

INSERT INTO t_forums(c_code,c_name,c_sequence)
values(
'SUPP','On-Line Support',2
);

CREATE TABLE t_messages(
c_forum_code VARCHAR(4) NOT NULL,
c_number INTEGER NOT NULL,
c_posted DATETIME NOT NULL,
c_subject VARCHAR(50) NOT NULL,
c_sender VARCHAR(32) NOT NULL,
c_message TEXT NOT NULL,
PRIMARY KEY(c_forum_code,c_number),
FOREIGN KEY(c_sender) REFERENCES t_users(c_uid)
);

```

) ;

Executing this script from MySQL is relatively easy. The procedure for executing the script and setting up MySQL is covered in [Appendix C](#).

Setting Up the Forum on Microsoft Access

The examples provided in this chapter will also work with the Microsoft Access database. To use our examples, you must have a copy of the database file forum.mdb. You can find this file with the other source code examples at <http://www.sams.com/>.

You should copy the forum.mdb file to your hard drive. If you own a copy of Microsoft Access, you can double-click forum.mdb to see the contents of the database.

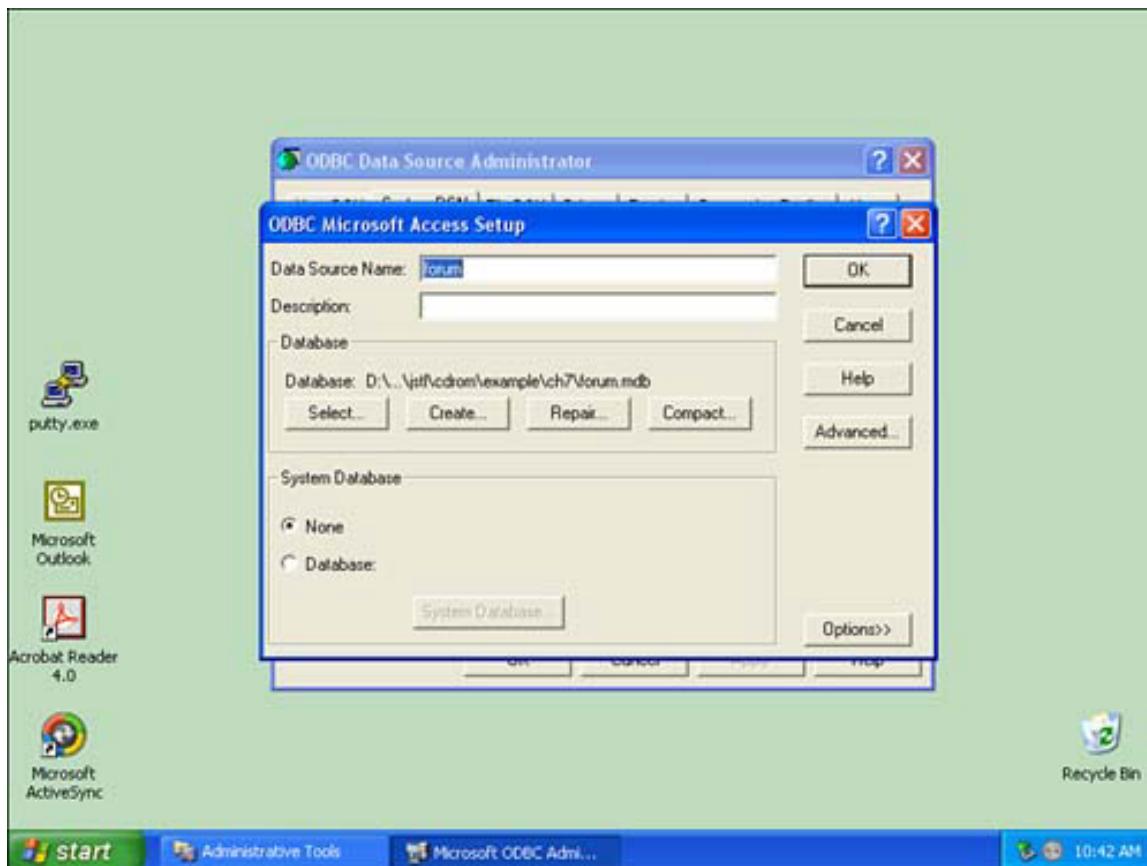
Once you install the file, you must create a data source name (DSN) for it. You do this in the Data Sources (ODBC) window. If you are using Windows XP, you'll find this item under Administrative Tools. If you are using an earlier copy of Windows, Data Sources (ODBC) is accessible from the main Control Panel.

At this point, double-click Data Sources (ODBC) to open the ODBC Data Source Administrator window. From here, you can configure new DSNs.

You must create a DSN for the forum application to work properly with Microsoft Access. This will allow JSTL to locate forum.mdb. Click the System DSN tab; you want to create a system-level DSN that will be accessible to every user of the operating system. Click the Add button and select the Microsoft Access Driver (*.mdb) option. Then, click OK to open the Microsoft Access ODBC configuration window.

There are only two pieces of information that you must supply. First, you must provide a name for your DSN. The DSN that you should use for our example is forum. Second, you must provide the location of the forum.mdb file. To do so, click the Select button on the Database tab. If you have properly configured your DSN, you should see the window shown in [Figure 7.2](#).

Figure 7.2. Configuring the DSN.

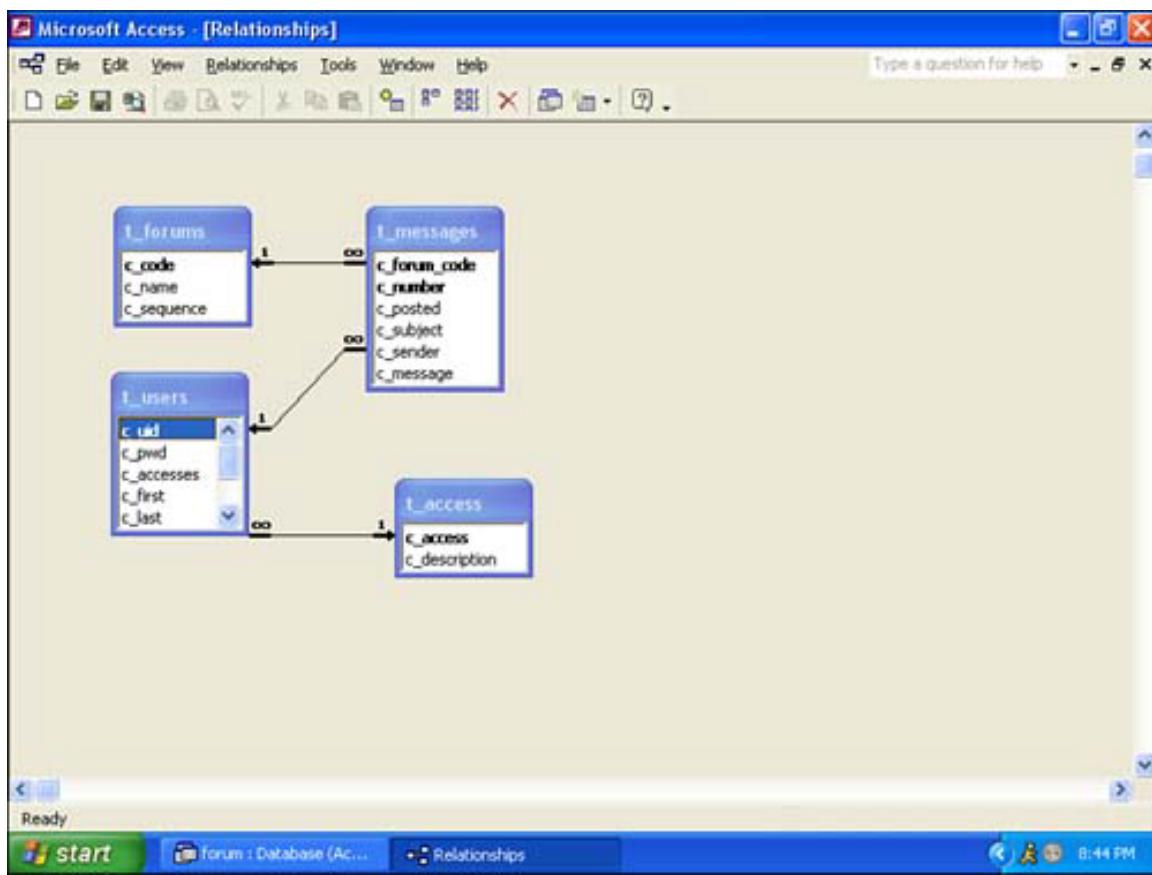


Database Tables

Your database should now be properly configured for either MySQL, Microsoft Access, or another database system. Now let's examine the structure of the database.

Figure 7.3 shows the Access Relationships window, which contains the layout of the database. This layout applies regardless of which database you are using.

Figure 7.3. Our database layout.



To ensure compatibility with the largest number of databases, you must follow certain conventions when naming tables and columns. For example, you cannot use a reserved word for the name of a column or table. It would be illegal, for instance, to have a table named select because this keyword is reserved for a commonly used SQL command. Certain databases will permit you to enclose the keyword in brackets to designate it as a column or table name. Unfortunately, support for this is not universal.

At this point, we'll describe the naming convention used for tables and columns in this book. All tables are prefixed with `t_`. For example, the access type table used by the forum application is named `t_access`, not just `access`. Column names are prefixed with `c_`. For example, if we need a column that stores a description, we use the name `c_description`. This allows us to be sure that the names used in our application are compatible with any database system, regardless of the reserved words imposed by that database.

Let's now examine the tables that make up our forum application; we begin with the access table. This table stores the different access modes, or user types, that the system allows. This table is summarized in [Table 7.1](#).

Table 7.1. The `t_access` Table

Column Name	Type	Length	Note
-------------	------	--------	------

c_access	VARCHAR	1	PK,NOT NULL
c_description	VARCHAR	32	NOT NULL

This table allows our forum application to store its three user types:

- Administrator, who is the super user and who can perform any action.
- Regular User, who can post and read messages but cannot perform any administrative actions.
- Guest, a user who has just registered. A guest may only read messages until his or her access has been upgraded by an administrative user.

As you can see in [Table 7.1](#), the t_access table has two columns. The first, c_access, contains the access code for the row. Of the three defined users, the administrators use A, the regular users use R, and the guests use G. The c_description column is used to specify the verbal user type, such as Administrator or Guest. This table is not modified as the application runs, since the number of user types is fixed.

The next table that we examine is the table that stores the user identification for the users who are registered to access the forum application. This table, named t_users, is shown in [Table 7.2](#).

Table 7.2. The t_users Table			
Column Name	Type	Length	Note
c_uid	VARCHAR	32	PK,NOT NULL
c_pwd	VARCHAR	32	NOT NULL
c_accesses	INTEGER		NOT NULL
c_first	DATETIME		NOT NULL
c_last	DATETIME		NOT NULL
c_bad	INTEGER		NOT NULL
c_posted	INTEGER		NOT NULL
c_type	VARCHAR	1	FK:t_access,NOT NULL

The primary key for this table is the c_uid column. This column specifies the user's login name, which, along with the password, is required to gain access to the system. The password is stored in the c_pwd column in an unencrypted format. For a production application, we suggest that you encrypt this password for added security. JSTL is not well suited to password encryption; password encryption is normally done in an EJB or custom tag library. We discuss password encryption in greater detail in [Chapter 11](#), "Creating Your Own Tag Libraries."

The c_type column contains a foreign key to the t_access table. Using this key, we are able to determine the user's type. This will become important later for security reasons.

The other columns in the t_users table simply store statistical information about the user. We won't use all of these columns in this chapter. We use the c_posted column in [Chapter 11](#) when we expand this application to use custom tag libraries.

Our application provides several forums that the user can choose to read or post messages to. The system must track the individual forums available. These forums are stored in the t_forums table, shown in [Table 7.3](#).

<i>Table 7.3. The t_forums Table</i>			
Column Name	Type	Length	Note
c_code	VARCHAR	4	PK,NOT NULL
c_name	VARCHAR	50	NOT NULL
c_sequence	INTEGER		NOT NULL

Three pieces of information are stored about a forum. The first is the forum code, which is the primary key for this forum; it should be a four-character abbreviation. The second is the name of the forum, which is stored in the c_name column. Finally, a sequence number is stored in c_sequence; this number allows the forums to be ordered. When the forums are displayed, they are sorted by sequence number. This allows the administrator to change the order in which the forums are displayed. This table contains no foreign keys to other tables.

The last table that we will examine is the table that stores the actual messages the users will read and post. This table, named t_messages, is shown in [Table 7.4](#).

<i>Table 7.4. The t_messages Table</i>			
Column Name	Type	Length	Note
c_forum_code	VARCHAR	4	PK,NOT NULL, FK:t_forums
c_number	INTEGER		PK,NOT NULL
c_posted	DATETIME		NOT NULL
c_subject	VARCHAR	50	NOT NULL
c_sender	VARCHAR	32	FK:t_users,NOT NULL
c_message	TEXT		NOT NULL

This table has a combined primary key that is made up of the c_forum_code and c_number columns. The c_forum_code key is also a foreign key to the t_forums table. The c_forum_code column designates which forum this message belongs to.

The c_number column contains a number assigned to individual messages within a forum. Messages are assigned increasingly higher numbers as they are posted to a forum. The person who posted this message is identified in the c_sender column, which is a foreign key to the t_users table.

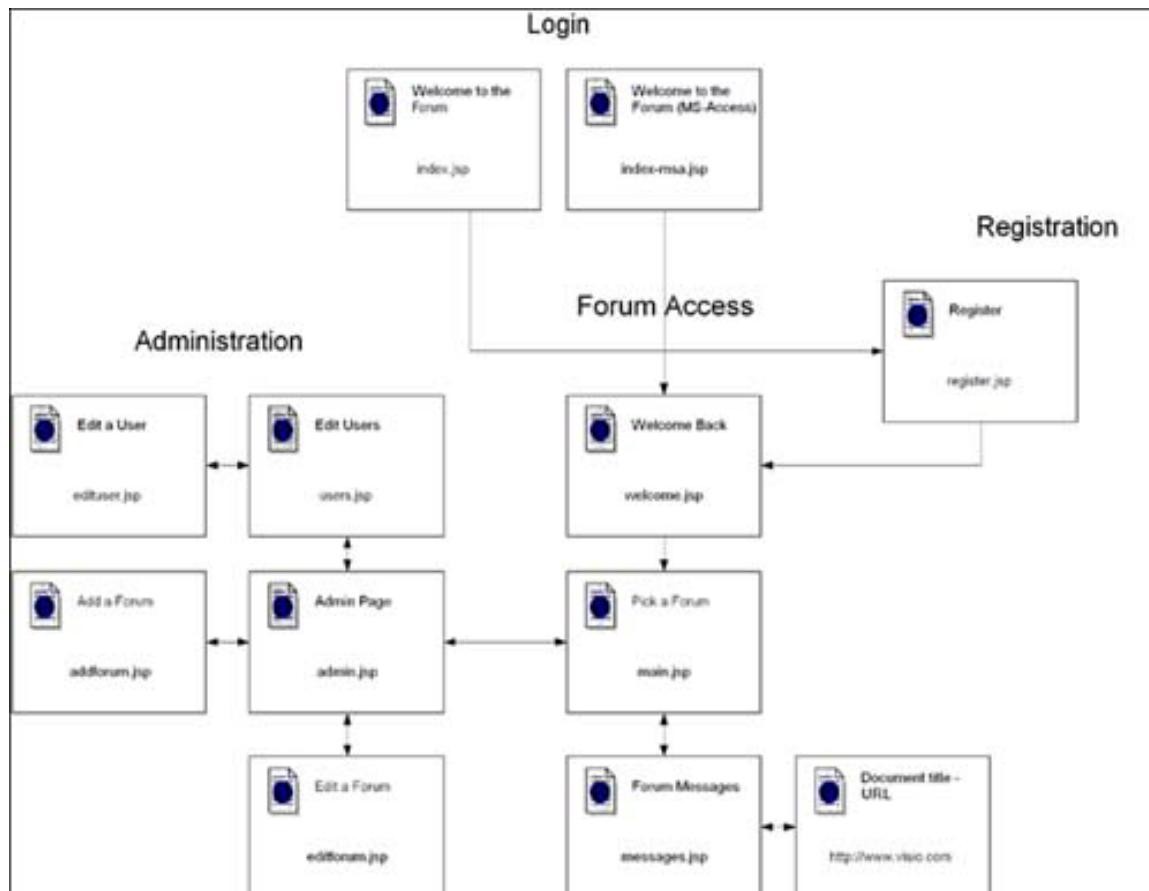
The actual text of the message is stored in the `c_message` column. An entry in this column should be defined as a text type that can hold larger amounts of data. In MySQL, you'd use the TEXT type. In Microsoft Access, you'd use the MEMO type.

The database holds all of the user information for our forum application. This database is used by a series of JSP pages that contain the JSTL that make up this application. Now that we have reviewed that database, let's examine the structure of the JSP pages that make up our forum application.

Screen Layout

Web applications consist of many Web pages that are interconnected through hyperlinks, posts, and other forms of Web navigation. [Figure 7.4](#) shows the flow from page to page in our forum application.

Figure 7.4. The screen flowchart for our forum application.



As you can see in [Figure 7.4](#), there are two entry points for the Web application:

- `index.jsp`, an index file that contains the driver used for MySQL
- `index-msa.jsp`, an index file that contains the driver used for Microsoft Access

Except for these two index files, all of the JSP files remain the same as those in the database programs. Only a few lines are different between index.jsp and index-msa.jsp. The difference is in how these files access the database. Let's now see how JDBC drivers are used to access the database.

Understanding JDBC Drivers

The SQL tags provided by JSTL make use of Java Database Connectivity (JDBC) drivers. Most commonly used databases provide a JDBC driver.

Sun also makes available a bridge driver that allows Java programs to make use of ODBC databases. Usually, you should try to find a suitable JDBC driver before you use the ODBC bridge driver. However, if no such driver exists, you may access the database using the ODBC bridge.

Connecting to a JDBC Data Source

To use the JSTL SQL tags, you have to identify a data source. Each of the JSTL SQL tags accepts a dataSource attribute that lets the SQL tag know what data source to use. Once the data source is specified, the JSTL tags will collaborate with that source to access the data requested.

JSTL provides three primary ways for setting up this collaboration. Let's take a look at each method.

Transparent Collaboration

For this approach, you must provide initialization code in the application logic of a server or other related Java code. For example, you could do this by using the application event listener of a server. The servlet must then store the application's default DataSource object in the javax.servlet.jsp.jstl.sql.dataSource application or session-scoped variable. This approach is advantageous because it makes the selection of the data source completely transparent to the page programmer. The JSP programmer does not need to specify a dataSource attribute to any of the JSTL tags being used, as shown here:

```
<sql:query ...>
```

Explicit Collaboration via Application Logic

It is also possible for servlet Java code to store a data source in a scoped variable for the page author to use. This variable must be stored at either session or application scope. You may use any scoped variable name. You must communicate the scoped variable name and intended scope to the JSP page programmer, who is then free to use this scoped variable in SQL tags. A typical SQL tag using this method would look like this:

```
<sql:query dataSource="\${dataSource}" ...>
```

Explicit Collaboration via the <sql:setDataSource> Tag

Both of the previous two approaches require access to more than just the JSP code. It is also possible to specify a data source using only JSTL tags from JSP. To use this method, you must use the `<sql:setDataSource>` tag to create a DataSource object implemented as a wrapper around JDBC's DriverManager class.

The following code loads the driver with the specified classname, and establishes a connection to the database associated with the given JDBC URL:

```
<sql:setDataSource var="dataSource"
driver="org.gjt.mm.mysql.Driver"
url="jdbc:mysql://localhost/forum?user=forumuser"/>
<sql:query dataSource="\${dataSource}" .../>
```

Using the <sql:setDataSource> Tag

You use the `<sql:setDataSource>` tag to create a data source that is contained in a scoped variable. You can use this tag in calls to the other JSTL SQL tags. The `<sql:setDataSource>` tag has one form:

```
<sql:setDataSource var="varName"
[scope="{page|request|session|application}"]
[driver="driverClassName"]
[url="jdbcUrl"]
[user="userName"] />
```

The `<sql:setDataSource>` tag accepts these attributes:

Attribute	Required	Purpose
<code>driver</code>	N	The classname of the JDBC data source that is to be used.
<code>scope</code>	N	The scope of the scoped variable specified by the attribute var. This attribute defaults to page.
<code>url</code>	N	The URL of the data source.
<code>user</code>	N	The user that is to be used to log into the data source.
<code>var</code>	N	The scoped variable that will hold the newly created data source.

As we mentioned earlier, you can use the `<sql:setDataSource>` tag to create ad hoc connections to a data source from JSP pages. This new data source will be assigned to the variable specified by the var attribute. You specify the scope of this variable with the scope attribute. If no scope is specified, then page scope is assumed.

JDBC requires two parameters to connect to a data source. First, you must specify the JDBC driver you are going to use with the driver attribute. Second, you must specify the

URL of the database that you are connecting to by using the url attribute. That URL is not the HTTP-type URL that you might be familiar with. The exact format of this URL is specified by the driver you are using. For more information on the format, consult the documentation associated with the driver you are using.

To show how to use the driver, we'll now look at the driver tag you must use to connect to MySQL and Microsoft Access. In the following sections, we assume that you have already set up either your Microsoft Access or your MySQL environment for the forum application. First, we'll show you how to connect to a MySQL data source.

Connecting to MySQL

The current version of MySQL does not come with a JDBC driver. To use MySQL with Java, you must obtain a driver elsewhere. One of the most common drivers used to allow Java to access the MySQL database is known as the MM driver. You can obtain this driver from <http://mssql.sourceforge.net/>.

Installing the driver is easy. You must copy the MM driver's JAR file—`mm.mysql-2.0.12-bin.jar`—to the `lib` directory of your Web application (most likely `C:\Program Files\Apache Tomcat 4.0\webapps\ROOT\WEB-INF\lib`). Once you've done this, you will be ready to use MySQL from your Web applications. If your Tomcat Web server was already running, you must restart it.

Now let's look at how you use the `setDataSource` tag to open a connection to the forum MySQL database. A Web page that performs a simple query is shown in [Listing 7.2](#).

Listing 7.2 Connecting to MySQL (query.jsp)

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

<sql:setDataSource var="dataSource"
driver="org.gjt.mm.mysql.Driver"
url="jdbc:mysql://localhost/forum?user=forumuser"/>

<html>
  <head>
    <title>Query Example</title>
  </head>

  <body>

<sql:query var = "users" dataSource="${dataSource}">
select c_uid,c_pwd,c_accesses,c_first,c_last,c_bad,c_posted,c_type
  from t_users
</sql:query>

<table border=1>
<c:forEach var="row" items="${users.rows}">
<tr>
```

```

<td><c:out value="${row.c_uid}" /></td>
<td><c:out value="${row.c_pwd}" /></td>
<td><c:out value="${row.c_accesses}" /></td>
<td><c:out value="${row.c_first}" /></td>
<td><c:out value="${row.c_last}" /></td>
<td><c:out value="${row.c_bad}" /></td>
<td><c:out value="${row.c_posted}" /></td>
<td><c:out value="${row.c_type}" /></td>
</tr>
</c:forEach>
</table>

</body>
</html>

```

The program in [Listing 7.2](#) begins by opening a connection to a MySQL database that resides on the local computer. The following setDataSource command accomplishes this:

```

<sql:setDataSource var="dataSource"
driver="org.gjt.mm.mysql.Driver"
url="jdbc:mysql://localhost/forum?user=forumuser"/>

```

If you want to connect to a MySQL database on a different computer, you must specify its hostname in place of localhost. Once you've executed this command, this means the scoped variable dataSource now contains a connection to the forum MySQL database. We did not specify a scope attribute, so this scoped variable will have page scope.

With the data connection open, you may now issue queries against it. This page performs a query against the t_users table and lists all registered users. You must specify the name of the data source in the `<sql:query>` tag so that the program finds the source:

```

<sql:query var = "users" dataSource="${dataSource}">
select c_uid,c_pwd,c_accesses,c_first,c_last,c_bad,c_posted,c_type from
t_users
</sql:query>

```

Now that we've seen how to use the `<sql:setDataSource>` tag with MySQL, let's see how to use it in Microsoft Access.

Connecting to Microsoft Access

It is not difficult to connect to a Microsoft Access database using JSTL. The `<sql:setDataSource>` tag must specify the ODBC bridge and a URL that will enable it to access a Microsoft Access database. We'll take the same basic query that we just used with MySQL and modify the JSP page so that it works with Access. This page is shown in [Listing 7.3](#).

Listing 7.3 Connecting to Microsoft Access

```

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

```

```

<%@ taglib uri="http://java.sun.com/jstl/sql" prefix="sql" %>

<sql:setDataSource var="dataSource"
driver="sun.jdbc.odbc.JdbcOdbcDriver"
url="jdbc:odbc:forum"/>

<html>
  <head>
    <title>Out Examples</title>
  </head>

  <body>

    <sql:query var = "users" dataSource="\${dataSource}">
      select c_uid,c_pwd,c_accesses,c_first,c_last,c_bad,c_posted,c_type
      from t_users
    </sql:query>

    <table border=1>
      <c:forEach var="row" items="\${users.rows}">
        <tr>
          <td><c:out value="\${row.c_uid}" /></td>
          <td><c:out value="\${row.c_pwd}" /></td>
          <td><c:out value="\${row.c_accesses}" /></td>
          <td><c:out value="\${row.c_first}" /></td>
          <td><c:out value="\${row.c_last}" /></td>
          <td><c:out value="\${row.c_bad}" /></td>
          <td><c:out value="\${row.c_posted}" /></td>
          <td><c:out value="\${row.c_type}" /></td>
        </tr>
      </c:forEach>
    </table>

  </body>
</html>

```

[Listing 7.3](#) is nearly identical to [Listing 7.2](#), with the exception of the driver tag. The driver tag used in [Listing 7.3](#) is designed to work with Access. Here's the setDataSource tag:

```

<sql:setDataSource var="dataSource"
driver="sun.jdbc.odbc.JdbcOdbcDriver"
url="jdbc:odbc:forum"/>

```

As you can see, the ODBC bridge driver is specified in the driver attribute. The URL that we've specified gives the DSN—forum—of the database we are trying to access. This driver tag will create a new driver in the scoped variable dataSource that holds the connection to the Access database. Once this connection is made, the remaining commands are the same ones used for the MySQL connection.

NOTE

There are some subtle differences between the syntax of SQL under MySQL and Microsoft Access. All of the sample SQL given in this chapter stays generic enough so that it will work on either. This is not always an easy task. If you want to ensure compatibility between two databases with the same SQL, you must test the SQL on both databases to be sure that it will work as you intend.

Connecting to Other Databases

Microsoft Access and MySQL are not the only databases that the JSTL SQL tags will work with. You can use any database that has a Java JDBC driver available. To use another database, you must find out the format of its driver and URL values. If you use these values correctly in the `<sql:setDataSource>` tag, then the database should be accessible to JSP pages.

Using SQL Tags

So far, we have examined the `<sql:setDataSource>` tag. JSTL makes available many other tags to facilitate the programming of SQL data sources. Let's examine each of these tags, beginning with the `<sql:query>` tag.

Using the `<sql:query>` Tag

The `<sql:query>` tag is used to perform a query of the database. A query is a SQL command that causes data to return. Not all SQL commands function in this way. The SELECT SQL command is most commonly used with the `<sql:query>` tag. If you are going to execute a SQL statement that does not return data—for example, INSERT, UPDATE, or DELETE—you must use the `<sql:update>` tag. The `<sql:query>` tag takes three forms:

```
// Syntax 1: Without body content
<sql:query sql="sqlQuery"
var="varName" [scope="{page|request|session|application}"]
[dataSource="dataSource"]
[maxRows="maxRows"]
[startRow="startRow"]/>
// Syntax 2: With a body to specify query arguments
<sql:query sql="sqlQuery"
var="varName" [scope="{page|request|session|application}"]
[dataSource="dataSource"]
[maxRows="maxRows"]
[startRow="startRow"]>
<sql:param> actions
</sql:query>
// Syntax 3: With a body to specify query and optional query
```

```

parameters
<sql:query var="varName"
[scope="{page|request|session|application}"]
[dataSource="dataSource"]
[maxRows="maxRows"]
[startRow="startRow"]>
query
optional <sql:param> actions
</sql:query>
```

The `<sql:query>` tag accepts the following attributes:

Attribute	Required	Purpose
<code>dataSource</code>	N	Specifies the data source to be used with this query.
<code>maxRows</code>	N	Specifies the maximum number of rows to be returned after startRow.
<code>scope</code>	N	Specifies the scope for the scoped variable specified by the var attribute. This attribute defaults to page scope.
<code>sql</code>	N	Specifies the SQL command that is to be executed.
<code>startRow</code>	N	Specifies the starting row that should be returned in the result set.
<code>var</code>	N	Specifies the scoped variable that will receive the results of this query.

The first syntax does not include a body. This allows you to specify a short SQL command as the sql attribute. If you are going to use longer SQL commands, it will be more practical to use the second syntax form, which lets you insert your SQL command into the body of the tag. This way, you can insert long SQL commands that include carriage returns to properly format the SQL.

The third syntax allows you to specify optional parameters that will be inserted into the SQL. Just as in JSTL, you may insert question marks (?) into your SQL. The question marks will be filled in with the parameter data that you specify. To learn more about the parameters, see the section about the `<c:param>` tag that appears later in this chapter.

You can control the number of rows returned by using the maxRows and startRow attributes. The maxRows attribute lets you specify the maximum number of rows that you would like returned. The startRow attribute lets you specify the row that you would like the results to begin with. Using these two attributes together, you can take subsections of data from a very large query.

The query tag will return a collection of the retrieved records. This collection will be stored in the scoped variable specified by the var attribute. The records returned by the query may be iterated by using the `<c:forEach>` tag. The following code demonstrates this:

```

<c:forEach var="row" items="${users.rows}">
<tr>
```

```

<c:out value="${row.c_uid}"/><br/>
</c:forEach>

```

This code displays the field named c_uid for every record in the collection users.

Using the <sql:update> Tag

Not all SQL commands return rows of data. The `<sql:update>` tag is used to invoke a SQL command that does not return data. Examples of SQL commands that do not return data are the UPDATE, DELETE, and INSERT commands. The `<sql:update>` tag has three forms:

```

// Syntax 1: Without body content
<sql:update sql="sqlUpdate"
[dataSource="DataSource"]
[var="varName"] [scope="{page|request|session|application}"]/>
// Syntax 2: With a body to specify update parameters
<sql:update sql="sqlUpdate"
[dataSource="DataSource"]
[var="varName"] [scope="{page|request|session|application}"]>
<sql:param> actions
</sql:update>
// Syntax 3: With a body to specify update statement and optional
// update parameters
<sql:update [dataSource="DataSource"]
[var="varName"] [scope="{page|request|session|application}"]>
update statement
optional <sql:param> actions
</sql:update>

```

The `<sql:update>` tag accepts these attributes:

Attribute	Required	Purpose
<code>dataSource</code>	N	Specifies the data source to be used with this command.
<code>scope</code>	N	Specifies the scope for the scoped variable specified by the var attribute. This attribute defaults to page scope.
<code>sql</code>	N	Specifies the SQL command that is to be executed.
<code>var</code>	N	Specifies the scoped variable that will receive the results of this command.

The first syntax does not include a body. This allows you to specify a short SQL command as the sql attribute. If you are going to use longer SQL commands, you should use the second syntax form, which lets you insert your SQL command into the body of the tag. This allows you to insert long SQL commands that may include carriage returns to properly format the SQL.

The third syntax allows you to specify optional parameters that will be inserted into the SQL. Just as in JSTL, you can insert question marks (?) into your SQL. The question

marks will be filled in with the parameter data that you specify. For more information about the parameters, see the section about the `<c:param>` tag that appears later in this chapter.

Just as with the `<sql:query>` tag, the `<sql:update>` tag returns a variable specified by the `var` attribute. This time, the variable is an Integer object, not a result set. This integer specifies the number of rows affected by the SQL command.

Using the `<sql:transaction>` Tag

Often, one SQL command cannot completely accomplish the task that you need. This can become a problem if one of the SQL commands fails to execute. Consider the example of two UPDATE commands that are used to move money from one account to another. The first UPDATE command inserts the money into the new account, and the second UPDATE command removes the money from the source account. If the second UPDATE command fails to execute yet the first one did, the user will be left with extra money. To alleviate this problem, you should use a transaction. This will cause both commands to fail if just one of the commands actually failed.

JSTL allows you to use transactions through the `<sql:transaction>` tag. You use this tag to specify a data source that you can use in calls to their SQL tags. There is one form of the `<sql:transaction>` tag:

```
<sql:transaction [dataSource="DataSource"]  
[isolation=isolationLevel]  
<sql:query> and <sql:update> statements  
</sql:transaction>  
isolationLevel ::= "read_committed"  
| "read_uncommitted"  
| "repeatable_read"  
| "Serializable"
```

The attributes accepted by the `<sql:transaction>` tag are as follows:

Attribute	Required	Purpose
<code>dataSource</code>	N	Specifies the data source to be used with this transaction.
<code>isolationLevel</code>	N	Specifies the isolation level for this transaction.

The body of the `<sql:transaction>` tag is used to specify other `<sql:update>` tags that perform the component commands of the transaction. You are given one parameter that allows you to specify the isolation level. These four transaction isolation levels are the same as the isolation levels specified to the JDBC Connection class:

- `TRANSACTION_READ_COMMITTED`— Dirty reads are prevented; nonrepeatable reads and phantom reads can occur.
- `TRANSACTION_READ_UNCOMMITTED`— Dirty reads, nonrepeatable reads, and phantom reads can occur.

- TRANSACTION_REPEATABLE_READ— Dirty reads and nonrepeatable reads are prevented; phantom reads can occur.
- TRANSACTION_SERIALIZABLE— Dirty reads, nonrepeatable reads, and phantom reads are prevented.

Using the <sql:param> Tag

We have already briefly mentioned that you can use parameters with your SQL commands. You use the `<sql:param>` tag to represent these parameters. There are two forms of the `<sql:param>` tag:

```
Syntax 1: Parameter value specified in attribute "value"
<sql:param value="value"/>
// Syntax 2: Parameter value specified in the body content
<sql:param>
parameter value
</sql:param>
```

The `<sql:dateParam>` accepts one parameter:

Attribute	Required	Purpose
value	N	The value to be inserted for this parameter.

The first syntax is the most common. In this form, you specify the value of the parameter by using the value attribute. The second syntax also lets you specify the value of this attribute in the body of the tag.

The order of the parameter tags in a `<sql:update>` or `<sql:query>` tag is important. The question mark (?) parameters specified in the SQL command are replaced in the same order as the `<sql:param>` tags are encountered. The `<sql:param>` tag should be used with all data types except dates. To specify a date, you need to use the `<sql:dateParam>` tag, as we explain next.

Using the <sql:dateParam> Tag

To specify date parameters, you should use the `<sql:dateParam>` tag. There are two forms of the `<sql:dateParam>` tag:

```
// Syntax 1: Parameter value specified in attribute "value"
<sql:dateParam value="value" type="[timestamp|time|date]" />
```

The `<sql:dateParam>` accepts two parameters:

Attribute	Required	Purpose
type	N	The type: timestamp, time, or date. The default value is timestamp.
value	N	The value to be inserted for this parameter.

You specify the value of this parameter by using the value attribute. The second syntax also lets you specify the value of and type of this attribute in the body of the tag. If the time type is specified, only a time will be used. If the date type is specified, only a date will be used. To use both, you should specify timestamp.

Implementing the Forum Example

You have now seen how the JSTL SQL tags work. You have also seen the basic screen and database structure of the forum application. We'll now show you how the forum application is implemented. This way, you can see all of the JSTL SQL tags in action.

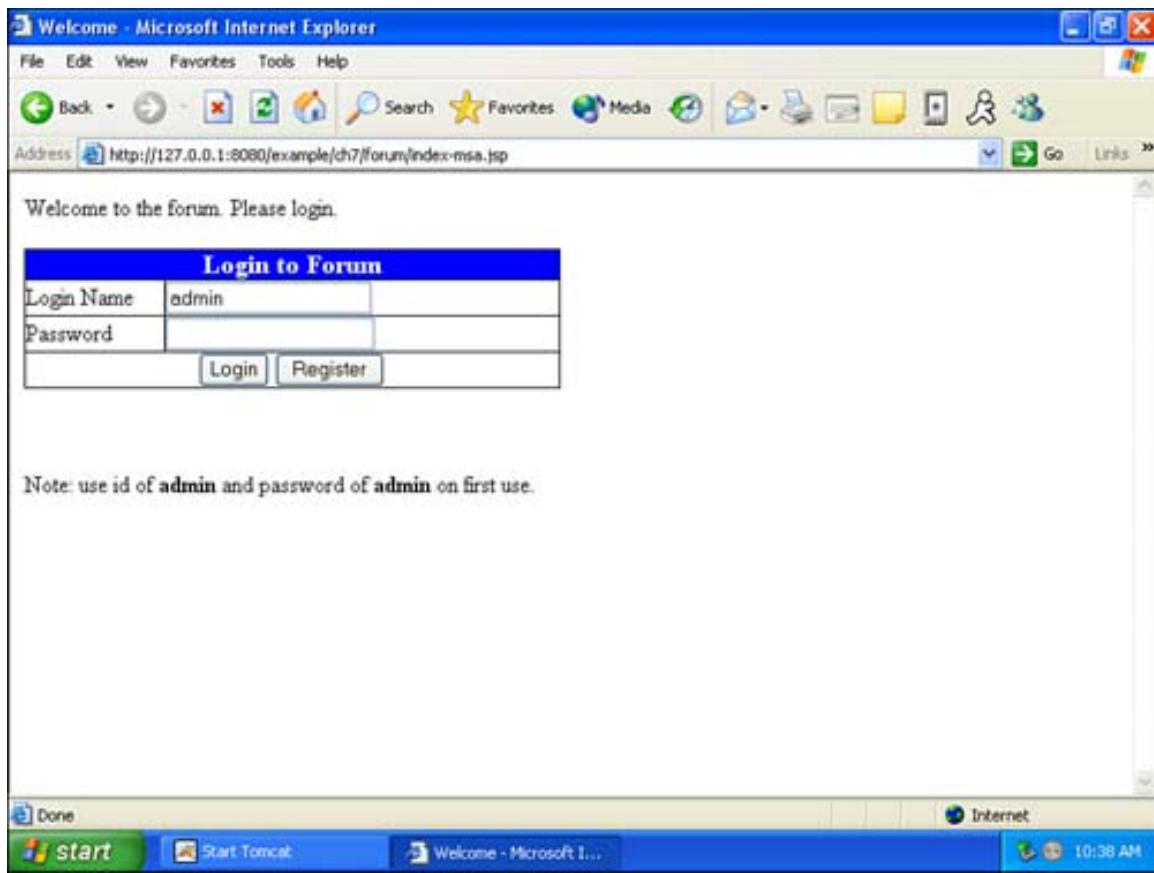
The listings in the following sections make up part of the forum application. We will examine only those listings that demonstrate unique uses of the JSTL SQL tags. You can find the source code to the complete forum application on the Sams Publishing Web site.

Let's start at the very beginning, with logging into the system.

Using a Query to Log In

The first page that a user will encounter on the forum application is index.jsp. This file displays a form that asks users either to enter their user ID and password, or to click the Register button to create a new account. [Listing 7.4](#) shows the source code to this page, and [Figure 7.5](#) shows what the form looks like.

Figure 7.5. The login page.



NOTE

[Listing 7.4](#) shows an example of an index file designed to work with MySQL. If you are using Microsoft Access, you should use the provided `index-msa.jsp` file. This file has been designed to work with Microsoft Access. All of the other files that make up the forum application will work with either database.

Listing 7.4 Logging In with MySQL (index.jsp)

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

<c:if test="${dataSource==null}">
    <sql:setDataSource var="dataSource" driver="org.gjt.mm.mysql.Driver"
        url="jdbc:mysql://localhost/forum?user=forumuser"
        scope="session" />
</c:if>

<c:if test="${pageContext.request.method=='POST' }">
    <c:if test="${param.reg!=null}">
        <c:redirect url="register.jsp" />
    </c:if>
```

```

<sql:query var="users" dataSource="${dataSource}">select c_uid,c_type
from t_users where c_uid = ? and c_pwd = ?
<sql:param value="${param.uid}" />

<sql:param value="${param.pwd}" />
</sql:query>

<c:choose>
    <c:when test="${users.rowCount<1}">
        <h3 color="red">Sorry, we have no one registered with that
        name.</h3>

        <sql:update var="result" dataSource="${dataSource}">update
        t_users set c_bad = c_bad + 1 where c_uid = ?
        <sql:param value="${param.uid}" />
        </sql:update>
    </c:when>

    <c:otherwise>
        <%
            Cookie mycookie = new
Cookie("login",request.getParameter("uid"));
            mycookie.setMaxAge(0x7fffffff);
            response.addCookie(mycookie);
        %>
        <c:forEach var="aUser" items="${users.rows}">
            <c:set var="userID" value="${aUser.c_uid}" scope="session" />

            <c:set var="userType" value="${aUser.c_type}" scope="session" />
        </c:forEach>

        <c:redirect url="welcome.jsp" />
    </c:otherwise>
</c:choose>
</c:if>

<c-rt:if test="<%=>request.getCookies() !=null%">
    <c-rt:forEach var="aCookie" items="<%=>request.getCookies()%>">
        <c:if test="${aCookie.name=='login'}">
            <c:set var="uid" value="${aCookie.value}" />
        </c:if>
    </c-rt:forEach>
</c-rt:if>

<html>
    <head>
        <title>Welcome</title>
    </head>

    <body>
        Welcome to the forum. Please login.<br>
        <form method="POST">
            <table border="1" cellpadding="0" cellspacing="0"
            style="border-collapse: collapse" bordercolor="#111111"
            width="49%" id="AutoNumber1">
                <tr>

```

```

<td width="100%" colspan="2" bgcolor="#0000FF">
    <p align="center">
        <b>
            <font color="#FFFFFF" size="4">Login to
            Forum</font>
        </b>
    </p>
</td>
</tr>

<tr>
    <td width="26%">Login Name</td>

    <td width="74%">
        <input type="text" name="uid" value=<c:out value=
        "${uid}" /> size="20" />
    </td>
</tr>

<tr>
    <td width="26%">Password</td>

    <td width="74%">
        <input type="password" name="pwd" size="20" />
    </td>
</tr>

<tr>
    <td width="100%" colspan="2">
        <p align="center">
            <input type="submit" value="Login" name="Login" />
            <input type="submit" value="Register" name="reg" />
        </p>
    </td>
</tr>
</table>

<p>&#160;</p>
</form>

<p>Note: use id of
<b>admin
</b> and password of
<b>admin
</b> on first use.
</p>
</body>
</html>
```

There are two distinct modes that the `index.jsp` page operates in. First, it must prepare to display the login form. Second, it must respond to the data posted by the login form. Let's first examine how the login form is prepared.

Preparing the Login Form

The first thing that the index page does is establish a connection with the database. To do this, the index page first checks to make sure that a connection is not already established. If there is no connection, then a new connection is opened. The following code does this:

```
<c:if test="${dataSource==null}">
    <sql:setDataSource var="dataSource" driver="org.gjt.mm.mysql.Driver"
        url="jdbc:mysql://localhost/forum?user=forumuser"
        scope="session" />
</c:if>
```

As you can see, the connection to the database is stored in the session variable dataSource. The connection will be maintained as long as the user's session remains active. This data source will be used through the application.

Next, the index.jsp file checks to see whether this request was a POST. If it was a POST, then we must process the results of the login form. In this section, we show you what to do if the request is not a POST.

The index page stores as a cookie the name of the user who last logged in. This allows the forum application to automatically default the user ID to the user ID of the last user who logged in. The following code attempts to locate the cookie that holds the ID of the last logged-in user:

```
<c-rt:if test="<%request.getCookies() != null%>">
    <c-rt:forEach var="aCookie" items="<%request.getCookies()%>">
        <c:if test="${aCookie.name=='login'}">
            <c:set var="uid" value="${aCookie.value}" />
        </c:if>
    </c-rt:forEach>
</c-rt:if>
```

This code uses RT tags to access the cookies collection. The cookies collection is then scanned for a cookie named login. If the login cookie is found, it will be used to set the default value for the login ID on the form.

Now, the index.jsp page is ready to display the login form. Once the form is displayed, the user can select to log in or register. When the user clicks one of these two buttons, the index.jsp page posts the data that the user entered back to itself. Let's now see what happens to the login screen when it receives a POST.

Handling a POST

The forum login page is designed to post back to itself. The following JSTL tag allows us to determine whether data has been posted:

```
<c:if test="${pageContext.request.method=='POST' }">
    <c:if test="${param.reg!=null}">
```

```
<c:redirect url="register.jsp" />
</c:if>
```

This code also checks to see whether the user clicked the Register button. Users are redirected to the register.jsp page if they click the Register button. If the user doesn't click Register, we must determine whether the user ID and password are correct. To do this, we construct a query that returns the rows that match both the user ID and password. There should be only one such row. The following code performs a query that checks to see whether there is a match:

```
<sql:query var="users" dataSource="${dataSource}">select c_uid,c_type
from t_users where c_uid = ? and c_pwd = ?
<sql:param value="${param.uid}" />
<sql:param value="${param.pwd}" />
</sql:query>
```

As you can see, this SQL command has two parameters and uses two `<sql:param>` tags. Now that the query has been executed, we must check the results. If no results are found, the following test becomes true:

```
<c:choose>
  <c:when test="${users.rowCount<1}">
    <h3 color="red">Sorry, we have no one registered with that
    name.</h3>
```

This means that the user login has failed. In this case, we should increment the bad login count for this user. The following UPDATE command does this. This command assumes that the user does exist, which has been verified by the previous `<c:when>` tag:

```
<sql:update var="result" dataSource="${dataSource}">update
t_users set c_bad = c_bad + 1 where c_uid = ?
<sql:param value="${param.uid}" />
</sql:update>
</c:when>
```

If the user record is found, then the user is valid. The following code executes when a valid login occurs:

```
<c:otherwise>
  <%
```

First, we need to set the cookie so that the user's name will default during the next login. The following code sets this cookie:

```
Cookie mycookie = new Cookie("login",request.
getParameter("uid"));
mycookie.setMaxAge(0x7fffffff);
response.addCookie(mycookie);
%>
```

Next, we examine the record that was returned. We copy the user's type and login name to session-scoped variables so that they can be accessed later:

```
<c:forEach var="aUser" items="${users.rows}">
    <c:set var="userID" value="${aUser.c_uid}" scope="session" />
    <c:set var="userType" value="${aUser.c_type}"
        scope="session" />
</c:forEach>
```

Now that the user has been logged in, we redirect the user to the welcome page. The following code does this:

```
<c:redirect url="welcome.jsp" />
```

In the next section, we examine what happens when the user registers.

Using an Update to Register

We saw how the `<sql:query>` tag can be used to check the user ID and password of a user. Now, let's see how to use the `<sql:update>` tag to add a new user to the database. Listing 7.5 shows the source code for the registration page, `register.jsp`, and Figure 7.6 shows our registration page.

Listing 7.5 Registering with the System (register.jsp)

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

<c:if test="${pageContext.request.method=='POST'}">
    <c:choose>
        <c:when test="${param.uid==''}">
            <h3>Must enter a login id!</h3>
        </c:when>

        <c:when test="${param.pwd==''}">
            <h3>Must enter a password!</h3>
        </c:when>

        <c:when test="${param.pwd!=param.pwd2}">
            <h3>Passwords do not match!</h3>
        </c:when>

        <c:otherwise>
            <sql:query var="users" dataSource="${dataSource}">
                select c_uid from t_users where c_uid = ?
                <sql:param value="${param.uid}" />
            </sql:query>

            <c:choose>
```

```

<c:when test="${users.rowCount>0}">
    <h3>Someone already has that user id.</h3>
</c:when>

<c:otherwise>
    <sql:update var="users" dataSource="${dataSource}">insert
        into t_users(c_uid,c_pwd,c_accesses,c_first,c_last,c_bad,c_
posted,c_type) values (?,?,?,?,?,0,0,'G')
        <sql:param value="${param.uid}" />
        <sql:param value="${param.pwd}" />
    </sql:update>
    <c:set var="userID" value="${param.uid}" scope="session" />
    <c:set var="userType" value="G" scope="session" />
    <c:redirect url="welcome.jsp" />
</c:otherwise>
</c:choose>
</c:otherwise>
</c:choose>
</c:if>

<html>
    <head>
        <title>Register</title>
    </head>

    <body>
        <p>
            Thanks for joining the forum. Please choose a login name
            and a password. Please enter your password twice to verify.
        </p>
        <form method="POST">
            <table border="1" cellpadding="0" cellspacing="0"
                style="border-collapse: collapse" bordercolor="#111111"
                width="49%" id="AutoNumber1">
                <tr>
                    <td width="100%" colspan="2" bgcolor="#0000FF">
                        <p align="center">
                            <b>
                                <font color="#FFFFFF" size="4">Register</font>
                            </b>
                        </p>
                    </td>
                </tr>
                <tr>
                    <td width="26%">Login Name</td>
                    <td width="74%">
                        <input type="text" name="uid" size="20" />
                    </td>
                </tr>
            </table>
        </form>
    </body>
</html>

```

```

<tr>
    <td width="26%">Password</td>

    <td width="74%">
        <input type="password" name="pwd" size="20" />
    </td>
</tr>

<tr>
    <td width="26%">Verify Password</td>

    <td width="74%">
        <input type="password" name="pwd2" size="20" />
    </td>
</tr>

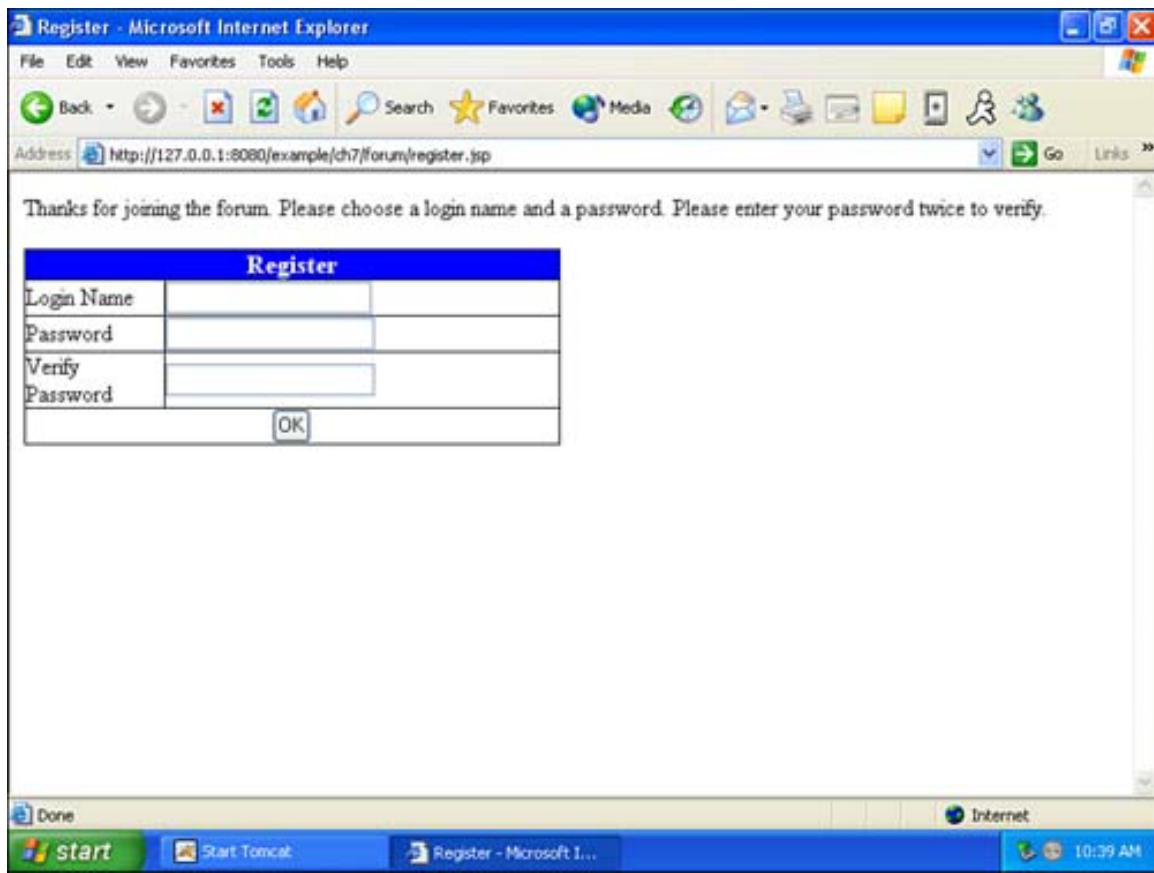
<tr>
    <td width="100%" colspan="2">
        <p align="center">
            <input type="submit" value="OK" name="OK" />
        </p>
    </td>
</tr>
</table>

<p>&#160;</p>
</form>

<p>&#160;</p>
</body>
</html>

```

Figure 7.6. The registration page.



The form displayed by this page is a basic HTML form with no data supplied. We begin by examining the process that the `register.jsp` page goes through when data is posted to it. As usual, the page begins with a check to determine whether this is a POST or simply the first time that the user is displaying the page:

```
<c:if test="${pageContext.request.method=='POST'}">
  <c:choose>
```

On this page, the user can log in as a new user. The structure of this page is similar to that of the login page. The page displays a form, as shown in [Figure 7.6](#), and asks the user to fill in information. When the user clicks the OK button, the data is posted back to the `register.jsp` page.

Once it has been determined that this is a POST, a `<c:choose>` tag block is entered that will determine whether the user failed to enter all required information. First, we check to see whether the user has entered a valid user ID:

```
<c:when test="${param.uid==''}">
  <h3>Must enter a login id!</h3>
</c:when>
```

Next, we check to see whether the user entered a password. The user is asked to enter the password again for verification. We do not need to check to see whether a second

password has been entered; if the second password is blank, the comparison between the second and the primary password will fail:

```
<c:when test="${param.pwd==' '}">
    <h3>Must enter a password!</h3>
</c:when>
```

Finally, we check to see whether the two passwords match. If they do, then we are ready to try to create an account for this user:

```
<c:when test="${param.pwd!=param.pwd2}">
    <h3>Passwords do not match!</h3>
</c:when>
```

Before we can create an account, we must ensure that the name chosen by the user has not already been taken. The following query checks for this:

```
<c:otherwise>
    <sql:query var="users" dataSource="${dataSource}">
        select c_uid from t_users where c_uid = ?
        <sql:param value="${param.uid}" />
    </sql:query>
```

If rows are returned, then we know that the user's ID has already been taken. If this is the case, we display an error and fall back through to the form so that the user must re-enter the user ID:

```
<c:choose>
    <c:when test="${users.rowCount>0}">
        <h3>Someone already has that user id.</h3>
    </c:when>
```

If the user's ID did match, we execute an insert statement that will add the user to the database:

```
<c:otherwise>
    <sql:update var="users" dataSource="${dataSource}">insert
        into t_users(c_uid,c_pwd,c_accesses,c_first,c_last,c_bad,c_
posted,c_type) values (?,?,?,?,NOW(),NOW(),0,0,'G')
        <sql:param value="${param.uid}" />
        <sql:param value="${param.pwd}" />
    </sql:update>
```

WARNING

Our query uses the `NOW()` function. This function works in Microsoft Access and MySQL, but may not work in other databases. If your database gives an error for this SQL command, you must replace the `NOW()` function with whatever function your database uses to return the current time and date.

The next step is to register two session variables that associate this session with the user who just logged in. To do this, we assign the session-scoped variable uid to the user ID. We also assign the session-scoped variable userType to the type of user who just logged on. The following code does this:

```
<c:set var="userID" value="${param.uid}"  
scope="session" />  
  
<c:set var="userType" value="G" scope="session" />  
  
<c:redirect url="welcome.jsp" />  
</c:otherwise>
```

Now we will remember what user is logged in from page to page. We can simply access the variable uid and determine the current user. We can also use the userType scoped variable to determine whether the user has sufficient security to undertake certain operations. As you can see, we set the user's type to G for Guest, since this is a new user. Now that the user has been registered, he or she will be taken to the welcome page—the same page that the login.jsp page goes to after a successful login.

Welcome to the Forum

We've seen how to query existing data and how to add new data to the database. In this section, we show you how to update data that already exists in the database.

When users successfully register or log in, they are taken to a welcome page. The source code to welcome.jsp is shown in [Listing 7.6](#).

Listing 7.6 Updating User Statistics (welcome.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>  
<%@ taglib uri="http://java.sun.com/jstl/core-rt" prefix="c-rt" %>  
<%@ taglib uri="http://java.sun.com/jstl/sql" prefix="sql" %>  
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt" %>  
<%@ taglib uri="http://java.sun.com/jstl/fmt-rt" prefix="fmt-rt" %>  
<c:if test="${ userID==null }">  
    <c:redirect url="index.jsp" />  
</c:if>  
<html>  
    <head>  
        <title>welcome</title>  
    </head>  
  
    <body link="#000000" vlink="#000000" alink="#000000">  
        <sql:query var="users" dataSource="${dataSource}">  
            select c_uid,c_pwd,c_accesses,c_first,c_last,c_bad,c_posted,c_  
            type from t_users where c_uid = ?  
            <sql:param value="${ userID}" />  
        </sql:query>  
  
        <c:forEach var="row" items="${users.rows}">
```

```

<table border="0" style="border-collapse: collapse"
bordercolor="#111111" cellpadding="0" cellspacing="0">
<tbody>
<tr>
<td bgcolor="#0000FF" colspan="2">
<b>
<font color="#FFFFFF">Welcome
<c:out value="${ userID}" />
</font>
</b>

<font color="#FFFFFF">&#160;</font>
</td>
</tr>

<tr>
<td bgcolor="#FFFF66"><b>Login ID</b></td>

<td bgcolor="#FFFF66">
<c:out value="${row.c_uid}" />
</td>
</tr>

<tr>
<td bgcolor="#FFFFFF"><b>Times on</b></td>

<td>
<c:out value="${row.c_accesses}" />
</td>
</tr>

<tr>
<td bgcolor="#FFFF66"><b>First Login</b></td>

<td bgcolor="#FFFF66">
<fmt:formatDate value="${row.c_first}" />
</td>
</tr>

<tr>
<td bgcolor="#FFFFFF"><b>Last Login</b></td>

<td>
<fmt:formatDate value="${row.c_last}" />
</td>
</tr>

<tr>
<td bgcolor="#FFFF66"><b>Failed Logins</b></td>

<td bgcolor="#FFFF66">
<c:out value="${row.c_bad}" />
</td>
</tr>

<tr>
<td><b>Messages Posted&nbsp;&nbsp;</b></td>

```

```

<td>
    <c:out value="${row.c_posted}" />
</td>
</tr>

<tr>
    <td bordercolor="#000000" bgcolor="#FFFF66"><b>User
Type</b></td>

    <td bgcolor="#FFFF66">
        <c:out value="${row.c_type}" />
    </td>
</tr>
</tbody>
</table>
</c:forEach>

<sql:update var="result" dataSource="${dataSource}">update
t_users set c_accesses = c_accesses + 1, c_bad = 0, c_last =
now() where c_uid = ?
<sql:param value="${userID}" />
</sql:update>

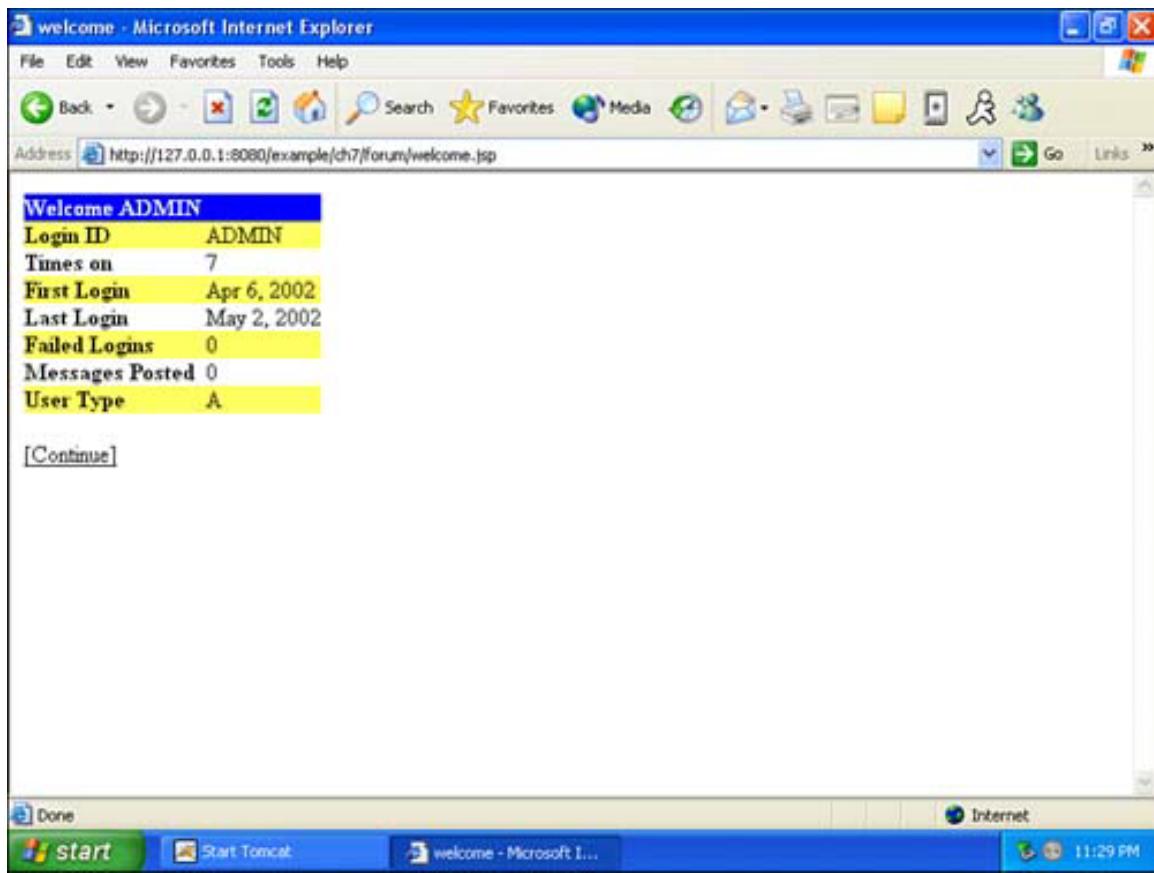
<br />

<a href="main.jsp">[Continue]</a>
</body>
</html>

```

The welcome page accomplishes two primary functions. First, it updates the user's statistics. The number of logins is increased by one, and the bad logins counter is reset to 0. The current date is inserted into the last successful login field. Second, this page displays the user's current statistics. The output from `welcome.jsp` is shown in [Figure 7.7](#).

Figure 7.7. The welcome page.



The JSTL code in welcome.jsp begins with a quick security check. The following code ensures that the user's session is valid:

```
<c:if test="${ userID==null }">
  <c:redirect url="index.jsp" />
</c:if>
```

If you examine any of the other source files, you'll notice that they contain this same check. The only source files that don't are index.jsp and register.jsp, since at that point the user has not yet logged on. If the user has reached one of the internal pages protected by the `<c:if>` tag and the user ID is not valid anymore, this means one of two things. Most likely, the user's session has expired, which happens after 30 minutes of inactivity by default. A second possibility is that a hacker is trying to access the site by skipping the login page. In either case, the user is simply routed back to the login page.

The next thing welcome.jsp does is update the user's statistics with this update statement:

```
<sql:update var="result" dataSource="${dataSource}">update
t_users set c_accesses = c_accesses + 1, c_bad = 0, c_last =
now() where c_uid = ?
<sql:param value="${userID}" />
</sql:update>
```

This code updates the user who has just logged in.

Using Transactions to Administer the Forum

Let's now see how the forum application uses the `<sql:transaction>` tags to properly delete records from the table. This is done in the main administration page at admin.jsp, shown in Listing 7.7.

Listing 7.7 Administration (admin.jsp)

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

<c:if test="${ userID==null }">
    <c:redirect url="index.jsp" />
</c:if>

<c:if test="${ userType!='A' }">
    <c:redirect url="main.jsp" />
</c:if>

<c:if test="${param.mode=='del'}">
    <sql:transaction dataSource="${dataSource}">
        <sql:update>
            delete from t_forums where c_code = ?
            <sql:param value="${param.target}" />
        </sql:update>

        <sql:update>
            delete from t_messages where c_forum_code = ?
            <sql:param value="${param.target}" />
        </sql:update>
    </sql:transaction>
    <h3>Forum deleted</h3>
</c:if>

<html>
    <head>
        <title>Admin</title>
    </head>

    <body link="#000000" vlink="#000000" alink="#000000">
        <sql:query var="forums" dataSource="${dataSource}">select
            c_code, c_name from t_forums order by c_sequence</sql:query>
        <p>From this screen you can edit forums, delete forums or create
            new forums.
        <br />
        You may also edit users.</p>
    </body>
</html>
```

```

<table border="0" width="393">
  <tbody>
    <tr>
      <td colspan="3" bgcolor="#0000FF" width="381">
        <b>
          <font size="4" color="#FFFFFF">The following forums are
          available:</font>
        </b>
      </td>
    </tr>

    <c:forEach var="row" items="${forums.rows}"
    varStatus="status">
      <jsp:useBean id="status"
      type="javax.servlet.jsp.jstl.core.LoopTagStatus" />

      <c-rt:choose>
        <c-rt:when test="<%==status.getCount()%2==0%">
          <c:set var="color" value="#FFFFFF" />
        </c-rt:when>

        <c-rt:otherwise>
          <c:set var="color" value="#FFFF66" />
        </c-rt:otherwise>
      </c-rt:choose>

      <c:url value="${pageContext.request.requestURI}" var="del">
        <c:param name="mode" value="del" />

        <c:param name="target" value="${row.c_code}" />
      </c:url>

      <c:url value="editforum.jsp" var="edit">
        <c:param name="mode" value="edit" />

        <c:param name="target" value="${row.c_code}" />
      </c:url>

      <tr bgcolor=<c:out value="${color}" />>
      <td width="192">
        <a href=<c:out value="${del}" />>[Delete]</a>

        <a href=<c:out value="${edit}" />>[Edit]</a>
      </td>

      <td width="22">
        <c:out value="${row.c_code}" />
      </td>

      <td width="165">
        <c:out value="${row.c_name}" />
      </td>
    </tr>
  </c:forEach>
</tbody>
</table>

```

```

    &#160;
<p>
    <a href="newforum.jsp">[New Forum]</a>

    <a href="users.jsp">[Edit Users]</a>

    <a href="main.jsp">[Exit]</a>
</p>
</body>
</html>

```

The administration page does the usual checks to see whether the user's session has expired. However, the administration page also ensures that the user has sufficient access rights to get to this page. If users reach this page without admin security, they are simply redirected to the main forum page, `main.jsp`:

```

<c:if test="${ userID==null }">
    <c:redirect url="index.jsp" />
</c:if>

<c:if test="${ userType!='A' }">
    <c:redirect url="main.jsp" />
</c:if>

```

This page is also responsible for handling forum deletes. To delete a forum, the page first checks to see whether the user has clicked one of the delete hyperlinks:

```
<c:if test="${param.mode=='del'}">
```

If the user has requested a delete, then a transaction is begun on the data source:

```
<sql:transaction dataSource="${dataSource}">
```

Deleting a forum involves two operations, and we need to ensure that both are carried out. If one or the other fails, we want both to fail. To do this, we use a transaction. The first component of the transaction deletes the forum from the `t_forums` table:

```

<sql:update>
    delete from t_forums where c_code = ?
<sql:param value="${param.target}" />
</sql:update>

```

In addition, we must ensure that the forum messages are deleted from the `t_messages` table:

```

<sql:update>
    delete from t_messages where c_forum_code = ?
<sql:param value="${param.target}" />
</sql:update>
</sql:transaction>

```

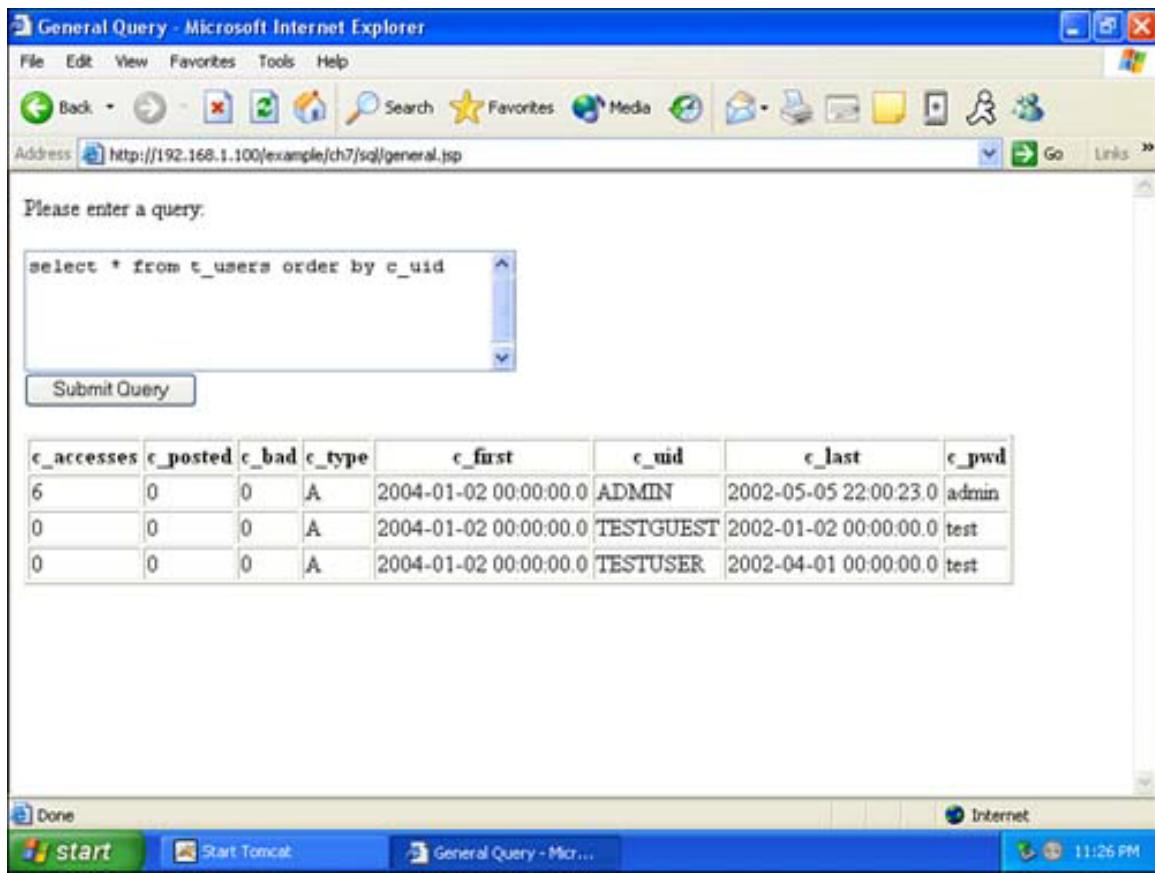
If both of these SQL operations can be carried out, then we consider the operation a success.

We have now seen how the forum example can be implemented using only JSTL tags. This approach is good for a small, or prototype, application. For larger applications, you should move all database access away from the JSP pages and use JSP for display only. In the next few chapters, we bring this program to that level. [Chapter 10](#), "Understanding JSTL Internationalization," will add multilingual abilities to our program. [Chapter 11](#) will show you how to use your own tag library to perform all database access and leave presentation to the JSP.

A General Query Engine

Let's look at one more SQL example. In every query that we've examined so far, we always knew exactly what columns were going to be returned. It is also possible to write your JSTL so that the results of a query will be displayed, regardless of what the names of the columns are. Our next example implements a general query form. Using this form, users can enter any SQL query that they like and see the results of the query displayed. [Figure 7.8](#) shows this program executing a query.

Figure 7.8. A general query page.



This program works in the same way as many of the other pages we've examined. It presents a form that posts back to the `general.jsp` page. Listing 7.8 shows the source code.

Listing 7.8 A General Query (general.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jstl/core-rt" prefix="c-rt" %>
<%@ taglib uri="http://java.sun.com/jstl/sql" prefix="sql" %>
<sql:setDataSource var="dataSource" driver="org.gjt.mm.mysql.Driver"
url="jdbc:mysql://localhost/forum?user=forumuser" />

<html>
  <head>
    <title>General Query</title>
  </head>

  <body>
    <c:choose>
      <c:when test="${param.cmd!=null}">
        <c:set var="str" value="${param.cmd}" />
      </c:when>

      <c:otherwise>
        <c:set var="str">
```

```

        value="select * from t_users" />
    </c:otherwise>
</c:choose>

Please enter a query:
<br />

<form method="post">
    <textarea name="cmd" cols="40" rows="5">
<c:out value="${str}" />
    </textarea>

    <br />

    <input type="submit" />
</form>

<c:if test="${pageContext.request.method=='POST' }">
    <c:catch var="e">
        <sql:query var="users" dataSource="${dataSource}"
        sql="${param.cmd}" />

        <table border="1">
            <c:forEach var="row" items="${users.rows}"
            varStatus="status">
                <jsp:useBean id="status"
                type="javax.servlet.jsp.jstl.core.LoopTagStatus" />

                <c-rt:if test="<%!=status.getCount ()==1%>">
                    <tr>
                        <c:forEach var="col" items="${row}">
                            <th>
                                <c:out value="${col.key}" />
                            </th>
                        </c:forEach>
                    </tr>
                </c-rt:if>

                    <tr>
                        <c:forEach var="col" items="${row}">
                            <td>
                                <c:out value="${col.value}" />
                            </td>
                        </c:forEach>
                    </tr>
                </c:forEach>
            </table>
    </c:catch>

    <c:if test="${e!=null}">
        <h3>Error</h3>

        <c:out value="${e}" />
    </c:if>
</c:if>
</body>
</html>

```

Like our previous examples, this program first checks to see whether it is being called with a POST. If so, it prepares to execute the query that the user specified:

```
<c:if test="${pageContext.request.method=='POST'}">
```

It is quite possible that the user might enter an invalid SQL command. To account for this, we use a `<c:catch>` tag that will catch these exceptions and allow us to handle them:

```
<c:catch var="e">
```

We now execute a query based on what the user entered. For this query, we use a bodyless `<sql:query>` tag. For the sql attribute, we specify the variable `param.cmd`, which holds the SQL command entered by the user:

```
<sql:query var="users" dataSource="${dataSource}"  
sql="${param.cmd}" />
```

The first thing that we must do is obtain the names of the columns. These names will be used as the column headers for the table columns when the results are displayed. To do this, we first begin iterating through the returned rows:

```
<table border="1">  
  <c:forEach var="row" items="${users.rows}"  
  varStatus="status">  
    <jsp:useBean id="status"  
    type="javax.servlet.jsp.jstl.core.LoopTagStatus" />
```

We now use the row status to determine which row number we are on. If we are on the first row, we display the column heads. The column heads are displayed by iterating through the `row` variable returned from the query. Then, we simply display the `key` attribute in each column:

```
<c-rt:if test="<%=(status.getCount ()==1)%>">  
  <tr>  
    <c:forEach var="col" items="${row}">  
      <th>  
        <c:out value="${col.key}" />  
      </th>  
    </c:forEach>  
  </tr>  
</c-rt:if>
```

The `key` attribute is not the only attribute available to iterate through in a row; we can also display all the values. For the other rows, we do that by using this code:

```
<tr>  
  <c:forEach var="col" items="${row}">  
    <td>  
      <c:out value="${col.value}" />  
    </td>  
  </c:forEach>
```

As you can see, the returned rows are collections themselves. By iterating through these rows, we are able to gain specific information about the structure of the query that we requested.

Summary

In this chapter, you learned how to use SQL tags from JSTL. Using SQL tags allows you to quickly construct JSTL-only applications. For production applications, you will likely isolate data access away from the JSP files; [Chapter 11](#) will show you how to do this.

Not all data is obtained using SQL. In the next chapter, we show you how to access XML data. We explain how you can use the SQL tags discussed in this chapter to both import and export XML data.

Chapter 8. Accessing Data with XML Tags

Considerable data is now available in the form of Extensible Markup Language (XML). Web pages often need to access XML data and display it. JSTL provides a range of XML tags that allow you to perform a variety of operations on XML data.

The JSTL tags enable you to iterate over XML data and perform comparisons on XML data using XPath expressions. You can also access individual elements of data within an XML document using XPath. In this way, you can customize the display of your XML using many of the JSTL tags you are already familiar with.

In addition, the JSTL tag library allows you to process data using Extensible Stylesheet Language templates (XSLT). By creating an XSL template, you can transform your XML data into HTML output, or even another XML document.

The XML tag library consists of three logical groups. The core tags perform the basic parsing and access to individual elements. Flow-control XML tags allow you to iterate over element collections and perform logical operations based on XPath expressions. Transformation operations allow you to use XSLT documents to reformat XML documents. We examine all three categories in this chapter. But first, let's look at XPath, a standard way of specifying sections of an XML document. The JSTL XML tag libraries make extensive use of XPath.

Understanding XML

An XML document is a hierarchical file consisting of nodes, attributes, and node values. Listing 8.1 shows a typical XML file. The examples presented in this chapter use this XML file.

Listing 8.1 A Typical XML File (students.xml)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<students>
    <student id="1">
        <name>
            <first>John</first>
            <last>Smith</last>
            <middle>T</middle>
        </name>
        <grade>
            <points>88</points>
            <letter>B</letter>
        </grade>
    </student>
    <student id="2">
        <name>
```

```

<first>James</first>
<last>Smith</last>
<middle>K</middle>
</name>
<grade>
    <points>92</points>
    <letter>A</letter>
</grade>
</student>
<student id="3">
    <name>
        <first>Kelly</first>
        <last>Lane</last>
        <middle>A</middle>
    </name>
    <grade>
        <points>72</points>
        <letter>C</letter>
    </grade>
</student>

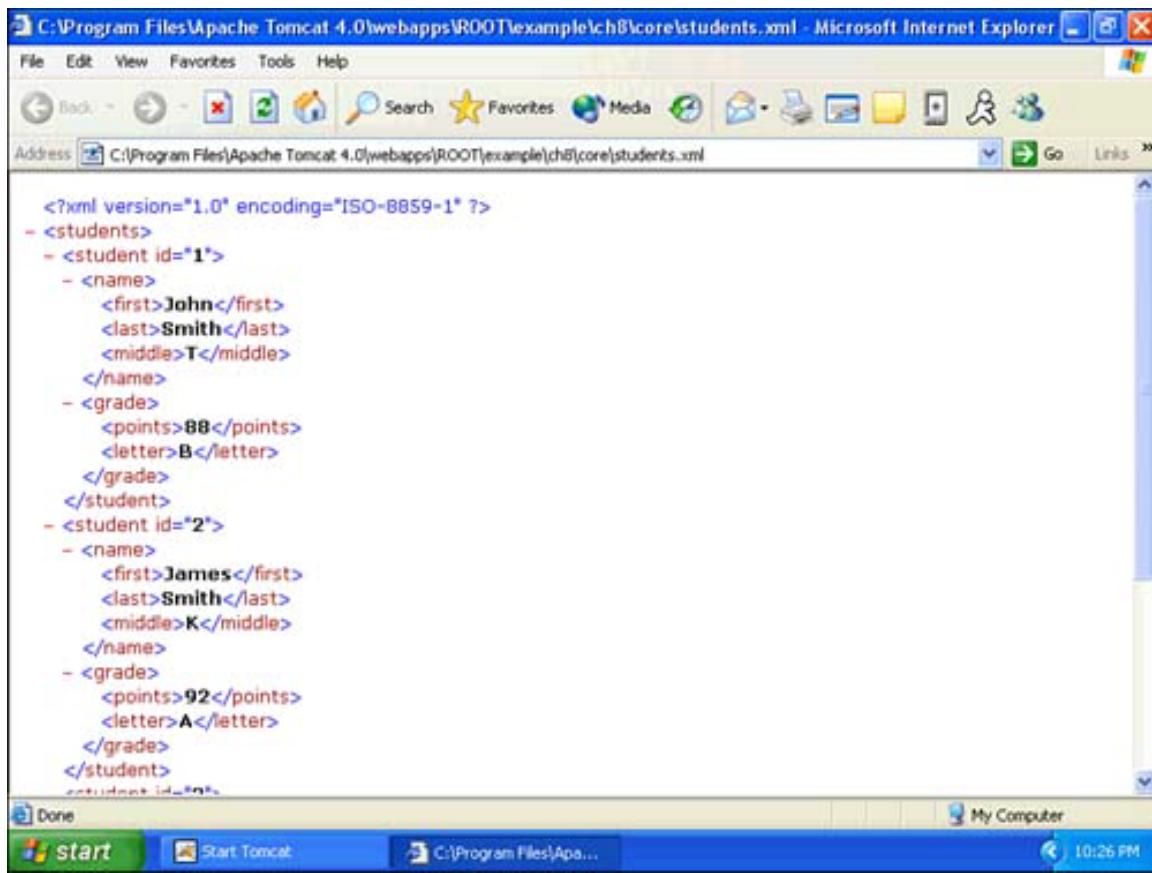
</students>

```

XML files must always have only one top-level node. As you can see in Listing 8.1, the top-level node for this file is the `<students>` tag. Inside the `<students>` node are additional nodes, named `<student>`, for each student. Each of the `<student>` nodes has an attribute called `id`. Inside each `<student>` node is a `<name>` node, and inside the `<name>` node are three more nodes: `<first>`, `<last>`, and `<middle>`. These three nodes have node values that contain the student's first name, last name, and middle initial, respectively.

If you are using Microsoft Windows or a Macintosh with Internet Explorer, you can use IE to view the XML file in hierarchical fashion. Simply double-click on the XML file to launch Internet Explorer and display the file, shown in Figure 8.1.

Figure 8.1. Viewing an XML file.



```
<?xml version="1.0" encoding="ISO-8859-1" ?>
- <students>
- <student id="1">
- <name>
<first>John</first>
<last>Smith</last>
<middle>T</middle>
</name>
- <grade>
<points>88</points>
<letter>B</letter>
</grade>
</student>
- <student id="2">
- <name>
<first>James</first>
<last>Smith</last>
<middle>K</middle>
</name>
- <grade>
<points>92</points>
<letter>A</letter>
</grade>
</student>
<student id="3">
```

By viewing the XML file in this way, you will be able to see its hierarchy. Using IE, you can easily expand and collapse branches of the XML file to focus on the data that you are interested in.

It is important to note that the XML standard does not specify how you should name nodes, attributes, or node values. XML is completely free-formed—which makes parsing XML challenging. One standard that attempts to help with this is XPath.

Next, we introduce the XML tags and learn how XPath is included. Let's begin by looking at the core XML tags used for basic parsing and access to individual data elements.

Understanding XPath

XPath is a sort of query language for XML. On an abstract level, XPath resembles SQL. Like SQL, XPath is a language for accessing data. XPath expressions are not executed by themselves; XPath is always used in conjunction with another programming language. Generally, XPath is used with XSLT, but other languages are now taking advantage of XPath expressions. JSTL is one of these languages.

Complete coverage of XPath is beyond the scope of this book; we provide only a brief introduction to XPath. For more information on XPath, you should refer to the World Wide Web Consortium (W3C) site at <http://www.w3.org/TR/xpath>.

XPath expressions are similar to paths used to reference files. For example, the XPath expression doc/students/student/name/first would access the first student's first name. The \$doc portion of the expression is provided by JSTL, and represents the actual XML document as the starting point. There are many other more complex forms of XSLT, some of which we examine in later chapters of this book.

Understanding XML Core Tags

Several core tags are provided by the JSTL XML tag library. These tags perform basic operations required by the other tags. The `<x:parse>` tag is used to parse XML data. When the `<x:parse>` tag is called, a variable is specified in which the parsed XML document will be stored. For example, consider the following code:

```
<!-- parse an XML document -->
<c:import url="http://www.site.com/students.xml" var="xml"/>
<x:parse source="${xml}" var="doc"/>
<!-- display using XPath expressions -->
<x:out select="$doc/name"/>
<!-- set a scoped variable -->
<x:set var="name" scope="request" select="$
doc/students/student/name/first"/>
```

This code begins by accessing the file <http://www.site.com/students.xml>. This file is loaded into the variable doc using the `<c:import>` tag. The `<c:import>` tag allows the contents of a URL to be downloaded into a scoped variable. This tag will be covered in greater detail in [Chapter 9](#), "Accessing Internet Resources with JSTL."

The contents of the downloaded XML file are then parsed using the `<x:parse>` tag. The resulting document is stored in the scoped variable doc.

Now that the document has been parsed, we can display some of the values. We do this by using XPath expressions. These expressions are specified as the select attribute that is passed to the `<x:out>` and `<x:set>` tags. The XML document is accessed by specifying its scoped variable as part of the XPath expression using the form
\$doc/students/student/name/first.

You will find that many of the JSTL XML tags use this form. You simply specify a select attribute that holds an XPath expression that you want to be evaluated.

Now that we have seen how the core XML tags work in general, let's examine each of these tags in detail. We begin with the `<x:parse>` tag.

Using the <x:parse> Tag

The `<x:parse>` tag parses XML data so that it may be accessed using the other XML JSTL tags. The two forms of the `<x:parse>` tag are shown here:

```
// Syntax 1: XML document specified via a String or Reader object
<x:parse {xml="XMLDocument"}
[filter="filter"]
[systemId="systemId"]
[{var="varName" [{scope="{page|request|session|application}"]
|varDom="varName"} [scopeDom="{page|request|session|application}"}]]/>
// Syntax 2: XML document specified via the body content
<x:parse [filter="filter"]
[systemId="systemId"]
[{var="varName" [{scope="{page|request|session|application}"
scopeDom="{page|request|session|application}"}]}>
    XML Document to parse
</x:parse>
```

The `<x:parse>` tag accepts the following attributes:

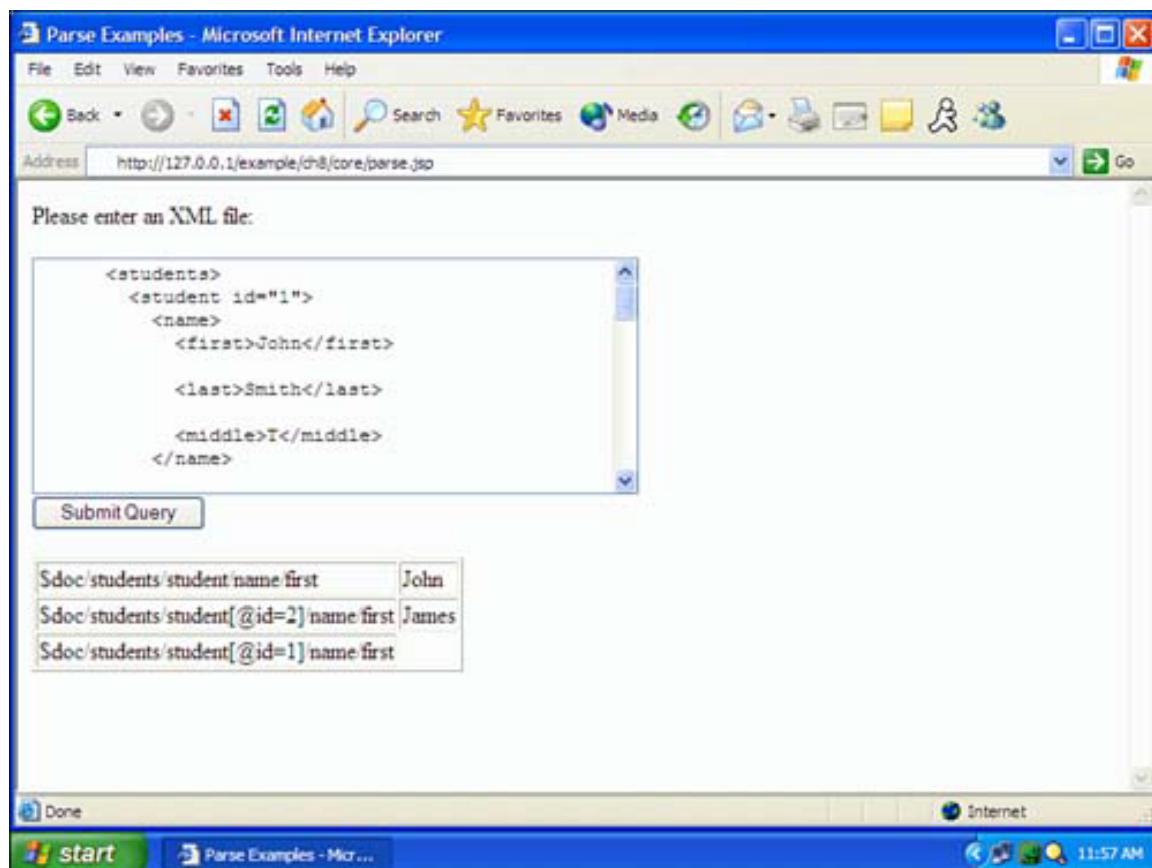
Attribute	Required	Purpose
<code>filter</code>	N	Allows a filter of the type <code>org.xml.sax.XMLFilter</code> to be used to filter the incoming XML data.
<code>scope</code>	N	Specifies the scope of the scoped variable referenced by the <code>var</code> attribute.
<code>scopeDom</code>	N	Specifies the scope of the scoped variable referenced by the <code>varDom</code> attribute.
<code>systemID</code>	N	Contains the URI for the parsing XML document.
<code>var</code>	N	Specifies the scoped variable referenced by the <code>varDom</code> attribute.
<code>varDom</code>	N	Specifies the scoped variable that will receive the XML document as an <code>org.w3c.dom.Document</code> object.
<code>xml</code>	N	Specifies the XML that is to be parsed.

The first syntax is used to parse XML data from a string, which is specified by the `xml` attribute. You may also take the XML directly from a URL by using the `xml` attribute. The second syntax allows you to specify XML data as the body of the tag. The XML document will be parsed and stored in the variable referenced by the `var` attribute. This variable will be used in conjunction with the other XML tag commands. In addition, you can specify a scope by using the `scope` attribute.

You may also obtain access to the newly parsed XML document as a `org.w3c.dom.Document` object. To obtain such an object, you have to specify a variable name by using the `varDom` attribute. You can also use `scopeDom` to specify a scope for this variable.

Let's look at an example that parses XML data. The sample parsing page displays a text box and allows you to paste XML data into the text box. Our application expects XML data in the form shown in Listing 8.1. It is permissible to have more records, but you may not change the structure of the records. When you click Submit Query, the program attempts to parse the data that you just provided and display the results. You can see this program in Figure 8.2.

Figure 8.2. Parsing XML data.



Let's now see how this program was constructed. The complete source code to this application appears in Listing 8.2.

Listing 8.2 Parsing XML (parse.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jstl/xml" prefix="x" %>
<html>
  <head>
    <title>Parse Examples</title>
  </head>

  <body>Please enter an XML file:
  <br />
```

```
<form method="post">
  <textarea rows="10" cols="50" name="xml">
    <students>
      <student id="1">
        <name>
          <first>John</first>

          <last>Smith</last>

          <middle>T</middle>
        </name>

        <grade>
          <points>88</points>

          <letter>B</letter>
        </grade>
      </student>

      <student id="2">
        <name>
          <first>James</first>

          <last>Smith</last>

          <middle>K</middle>
        </name>

        <grade>
          <points>92</points>

          <letter>A</letter>
        </grade>
      </student>

      <student id="3">
        <name>
          <first>Kelly</first>

          <last>Lane</last>

          <middle>A</middle>
        </name>

        <grade>
          <points>72</points>

          <letter>C</letter>
        </grade>
      </student>
    </students>
  </textarea>

  <br />

  <input type="submit" />
</form>
```

```

<c:if test="${pageContext.request.method=='POST' }">

    <x:parse var="doc" xml="${param.xml}" />

    <table border="1">
        <tr>
            <td>$doc/students/student/name/first</td>

            <td>
                <x:out select="$doc/students/student/name/first" />
            </td>
        </tr>

        <tr>
            <td>$doc/students/student[@id=2]/name/first</td>

            <td>
                <x:out
                    select="$doc/students/student[@id=2]/name/first" />
            </td>
        </tr>

        <tr>
            <td>$doc/students/student[@id=1]/name/first</td>

            <td>
                <x:out
                    select="$doc/students/student[@id=1]/name/first" />
            </td>
        </tr>
    </table>
</c:if>
</body>
</html>

```

The parsing example in Listing 8.2 operates in the two modes in which most of the form applications in this book operate. When the user first displays the page, our application displays only the form. The second half of the program does not begin until the user clicks the Submit Query button. This process begins by checking to see whether the current request was a POST, which indicates that the user has responded to the form. The following code checks to see whether the page was called as a result of the form being posted:

```
<c:if test="${pageContext.request.method=='POST' }">
```

Once we determine that the user has responded to the form, we can begin to parse the file. To begin, we call the `<x:parse>` tag, and pass in the XML data from the form as the `xml` attribute:

```
<x:parse var="doc" xml="${param.xml}" />
```

Now, we can begin to display the data. This `<x:out>` tag displays the first student's first name:

```
<x:out select="$doc/students/student/name/first" />
```

This is accomplished by using the XPath expression `doc/students/student/name/first`. Because we have several student records, this XPath statement must decide which student to use. Whenever a collection is presented to an XPath expression, the first element will be taken unless another element is specified.

XPath also allows us to get creative with our expressions. XPath can express many queries in XML data. The following `<x:out>` tag uses the searching feature of XPath. The XPath expression searches through all the student records and locates the student record with an attribute named `id` that holds the value 2. The following line of code does this:

```
<x:out select="$doc/students/student[@id=2]/name/first" />
```

This `<x:out>` statement used the XPath `$doc/students/student[@id=2]/name/first`. The `[@id=2]` suffix to the student identifier specifies that all student records should be scanned until one is located that has an attribute named `id` with the value 2.

Let's take a look at the `<x:out>` tag in greater detail.

Using the `<x:out>` Tag

You use the `<x:out>` tag to display data from a parsed XML document. There is one form of the `<x:out>` tag:

```
<x:out select="XpathExpression" [escapeXml="{true|false}"] />
```

The attributes accepted by the `<x:out>` tag are as follows:

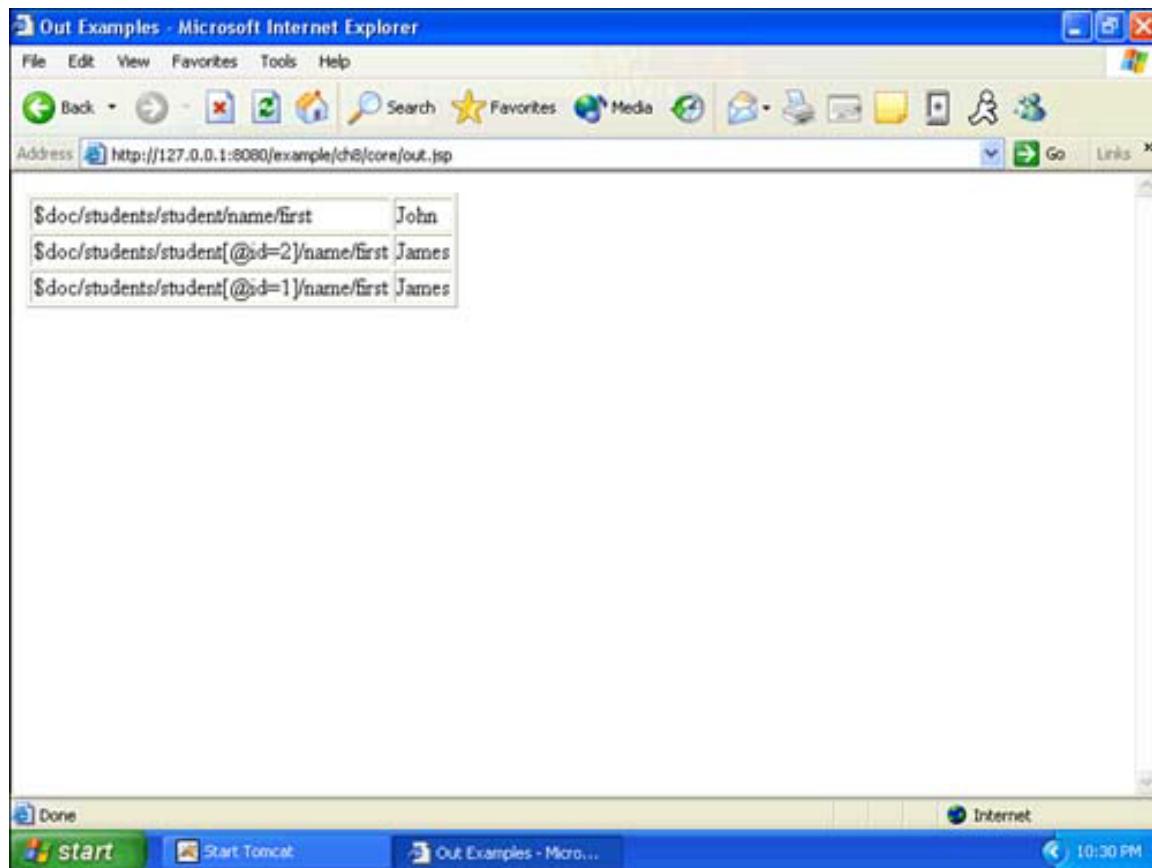
Attribute	Required	Purpose
<code>escapeXml</code>	N	Specifies whether the special characters of the data displayed should be escaped. The default value is true.
<code>select</code>	N	Specifies the XPath expression to be displayed.

This tag accepts an XPath expression from the `select` attribute. The expression will be displayed to the browser. You can select to escape the XML by using the `escapeXml` attribute. If you specify the value `true`, the characters `<,>,&,'`, and `"` will be converted to their character entry codes.

The `<x:out>` tag has the ability to display parts of the XML file. Let's look at another example. This program is similar to our earlier example, in which we retrieved the XML data from the page's form. In this example, we obtain the XML data from an external file.

We'll use the same file, students.xml, shown in [Listing 8.1](#). The output from this program appears in [Figure 8.3](#).

Figure 8.3. Using the <x:out> tag.



Let's discuss the differences between this page and the previous example. The Web page is shown in [Listing 8.3](#).

Listing 8.3 Using the Out Tag (out.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jstl/xml" prefix="x" %>
<html>
  <head>
    <title>Out Examples</title>
  </head>

  <body>
    <c:import var="students" url="students.xml" />

    <x:parse var="doc" xml="${students}" />

    <table border="1">
      <tr>
```

```

<td>$doc/students/student/name/first</td>

<td>
  <x:out select="$doc/students/student/name/first" />
</td>
</tr>

<tr>
  <td>$doc/students/student[@id=2]/name/first</td>

  <td>
    <x:out
      select="$doc/students/student[@id=2]/name/first" />
  </td>
</tr>

<tr>
  <td>$doc/students/student[@id=1]/name/first</td>

  <td>
    <x:out
      select="$doc/students/student[@id=1]/name/first" />
  </td>
</tr>
</table>
</body>
</html>

```

The process by which this example loads the XML data is considerably different from in the previous example. As we explained, this program loads the XML from an external file, which is stored on the same Web server as the sample program. This does not need to be the case; XSTL allows you to easily receive content from other Web sites, a concept we cover in [Chapter 9](#).

The following code loads the students.xml file for our sample program:

```

<c:import var="students" url="students.xml" />
<x:parse var="doc" xml="${students}" />

```

As you can see, the XML file is first loaded into a string called `students`. This string is fed to the XML parser with the `<x:parse>` tag. The data is then displayed with the same `<x:out>` tags we used in our previous example.

Using the `<x:set>` Tag

The `<x:set>` tag works much like the `<x:out>` tag, except that the output is sent to a variable rather than displayed. There is one form of the `<x:set>` tag:

```

<x:set select="XpathExpression"
var="varName" [scope="{page|request|session|application}"] />

```

The `<x:set>` tag accepts these attributes:

Attribute	Required	Purpose
scope	N	Specifies the scope for the scoped variable references by the var attribute. This attribute defaults to page scope.
select	N	Contains the XPath expression that you want to display.
var	Y	Specifies the scoped variable that you want to set.

You must specify the XPath expression using an attribute of the `<x:set>` tag. You specify the variable that you want to receive the results of the XPath expression by using the attribute var. You specify the variable's scope with the attribute scope. In addition, you have to specify the XPath expression that you would like evaluated by using the select attribute.

Let's look at an example that makes use of the `<x:set>` tag. This program closely follows our previous examples. However, rather than simply displaying the XML elements, this program copies them to scoped variables. These values are then displayed to the user. The end result is the same as shown in Figure 8.3. Listing 8.4 shows our program.

[Listing 8.4 Using the Set Tag](#)

```

<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jstl/xml" prefix="x" %>
<html>
  <head>
    <title>Set Examples</title>
  </head>

  <body>
    <c:import var="students" url="students.xml" />

    <x:parse var="doc" xml="${students}" />

    <x:set var="a" select="$doc/students/student/name/first" />

    <x:set var="b"
      select="$doc/students/student[@id=2]/name/first" />

    <x:set var="c"
      select="$doc/students/student[@id=1]/name/first" />

    <table border="1">
      <tr>
        <td>$doc/students/student/name/first</td>

        <td>
          <x:out select="$a" />
        </td>
      </tr>

      <tr>
        <td>$doc/students/student[@id=2]/name/first</td>

```

```

<td>
    <x:out select="$b" />
</td>
</tr>

<tr>
    <td>$doc/students/student[@id=1]/name/first</td>

    <td>
        <x:out select="$c" />
    </td>
</tr>
</table>
</body>
</html>

```

This program begins in the same way as the previous program, by loading the file `students.xml`:

```

<c:import var="students" url="students.xml" />
<x:parse var="doc" xml="${students}" />

```

Now that the XML file has been loaded, we can use the `<x:set>` tags to copy the values from their XML elements to the scoped variables a, b, and c:

```

<x:set var="a" select="$doc/students/student/name/first" />
<x:set var="b"
      select="$doc/students/student[@id=2]/name/first" />
<x:set var="c"
      select="$doc/students/student[@id=1]/name/first" />

```

Using XML Flow-Control Tags

The XML flow-control tags enable your program to make decisions based on the results of XSL queries. The XML flow-control tags resemble some of the JSTL core tags.

The `<c:if>` tag allows the core tag library to perform logic based on what's entered or stored in scoped variables. The XML tag library adds the tag `<x:if>`, which enables your program to conditionally execute other tags based on the result of an XPath expression. The `<c:if>` tag is not the only tag that has a counterpart in the XML tag library; the tags `<c:otherwise>`, `<c:choose>`, and `<c:when>` all have the XML counterparts `<x:otherwise>`, `<x:choose>`, and `<x:when>`.

Using the `<x:forEach>` Tag

The `<x:forEach>` tag works much like the `<c:forEach>` tag. Both tags allow you to iterate over a collection of values. As we discussed earlier, an XPath expression can

return a collection of nodes, and the `<x:forEach>` tag allows you to iterate over those nodes. There is one form of the `<x:forEach>` tag:

```
<x:forEach [var="varName"] select="XpathExpression">  
body content  
</x:forEach>
```

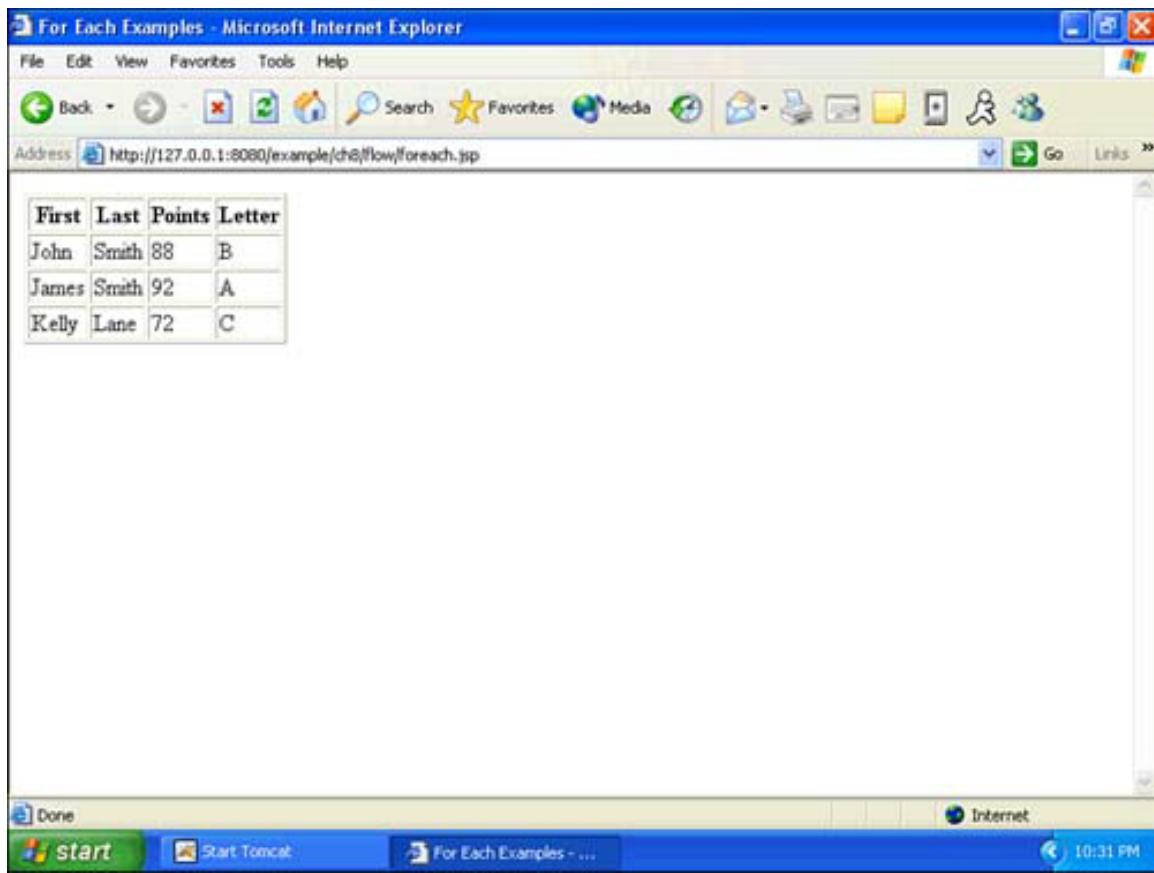
The `<x:forEach>` tag accepts the following attributes:

Attribute	Required	Purpose
<code>select</code>	N	An XPath expression that you want to iterate over.
<code>var</code>	Y	Specifies the scoped variable that will hold each iteration.

The value placed in the variable—which is specified by the `var` attribute—contains all the values stored in the node. You can use `<x:out>` and `<x:set>` with relative XPaths that will obtain data from each member of the collection.

So far, our examples have accessed only one student name at a time. Using the `<x:forEach>` tag, it is possible to iterate through entire branches of the XML file. Let's now look at an example that reads through the entire list of students and displays each student and his or her score. The output from this program is shown in [Figure 8.4](#).

Figure 8.4. Using the `<x:forEach>` tag.



Let's look at how this page is implemented. You'll find that using the `<x:forEach>` tag is similar to using the `<c:forEach>` tag that allows us to iterate over collections. Listing 8.5 shows how this page was constructed.

Listing 8.5 Using the <x:forEach> Tag (foreach.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jstl/xml" prefix="x" %>
<html>
  <head>
    <title>For Each Examples</title>
  </head>

  <body>
    <c:import var="students" url="students.xml" />

    <x:parse var="doc" xml="${students}" />

    <table border="1">
      <tr>
        <th>First</th>

        <th>Last</th>

        <th>Points</th>
```

```

<th>Letter</th>
</tr>

<x:forEach var="student" select="$doc/students/student">
  <tr>
    <td>
      <x:out select="name/first" />
    </td>

    <td>
      <x:out select="name/last" />
    </td>

    <td>
      <x:out select="grade/points" />
    </td>

    <td>
      <x:out select="grade/letter" />
    </td>
  </tr>
</x:forEach>
</table>
</body>
</html>

```

The program in Listing 8.5 begins by loading the student.xml file, just as the previous programs did:

```

<c:import var="students" url="students.xml" />
<x:parse var="doc" xml="${students}" />

```

The student list is displayed in an HTML table. The following code sets up the header for this table:

```

<table border="1">
  <tr>
    <th>First</th>
    <th>Last</th>
    <th>Points</th>
    <th>Letter</th>
  </tr>

```

At this point, we must create the individual rows that display each of the students. This is done with the following `<x:forEach>` tag, which iterates over all student nodes contained in the XML file:

```

<x:forEach var="student" select="$doc/students/student">

```

As the program iterates through the students branch of the XML file, it places each row in the student scoped variable we specified in the var attribute. As you can see, we're

using the XPath query \$doc/students/student. We use this code to display each student as a row in the table containing the first name, last name, grade points, and letter grade:

```
<tr>
  <td>
    <x:out select="name/first" />
  </td>
  <td>
    <x:out select="name/last" />
  </td>
  <td>
    <x:out select="grade/points" />
  </td>
  <td>
    <x:out select="grade/letter" />
  </td>
</tr>
```

Now we have only to end the `<x:forEach>` tag and the table:

```
</x:forEach>
</table>
```

Using the `<x:if>` Tag

You use the `<x:if>` tag to evaluate the result of an XPath expression. This expression should result in a Boolean value. There are two forms of the `<x:if>` tag:

```
// Syntax 1: Without a body, result copied to var
<x:if select="XpathExpression"
var="varName" [scope="{page|request|session|application}"]/>
// Syntax 2: With conditional body content
<x:if select="XpathExpression"
[var="varName"] [scope="{page|request|session|application}"]>
  body content
</x:if>
<x:if select="PathExpression"
[var="varName"] [scope="{page|request|session|application}"]>
  body content
</x:if>
```

The first syntax uses an attribute, var, to specify a scoped variable to hold the true/false result of the comparison. The second form uses the body of the `<x:if>` tag to specify a block of the document that will be executed conditionally. Here are the attributes accepted by the `<x:if>` tag:

Attribute	Required	Purpose
<code>scope</code>	N	If the var attribute is specified, the scope attribute specifies the scope of that variable. Defaults to page scope.
<code>select</code>	Y	The XPath expression that you want to evaluate.

<code>var</code>	N	A scoped variable that will receive the value of the expression being evaluated. This attribute is required if there is no body.
------------------	---	--

As with many of the JSTL tags, you must provide a select attribute that specifies the XPath expression that you want to evaluate. You may optionally include a var attribute that specifies a scoped variable to store the result of the comparison. This scoped variable will receive the Boolean value true or false. If you include a var attribute, you can also specify a scope attribute that specifies the variable's scope. If the XPath expression evaluates to true, then the body of the `<x:if>` tag is executed.

Using the `<x:choose>` Tag

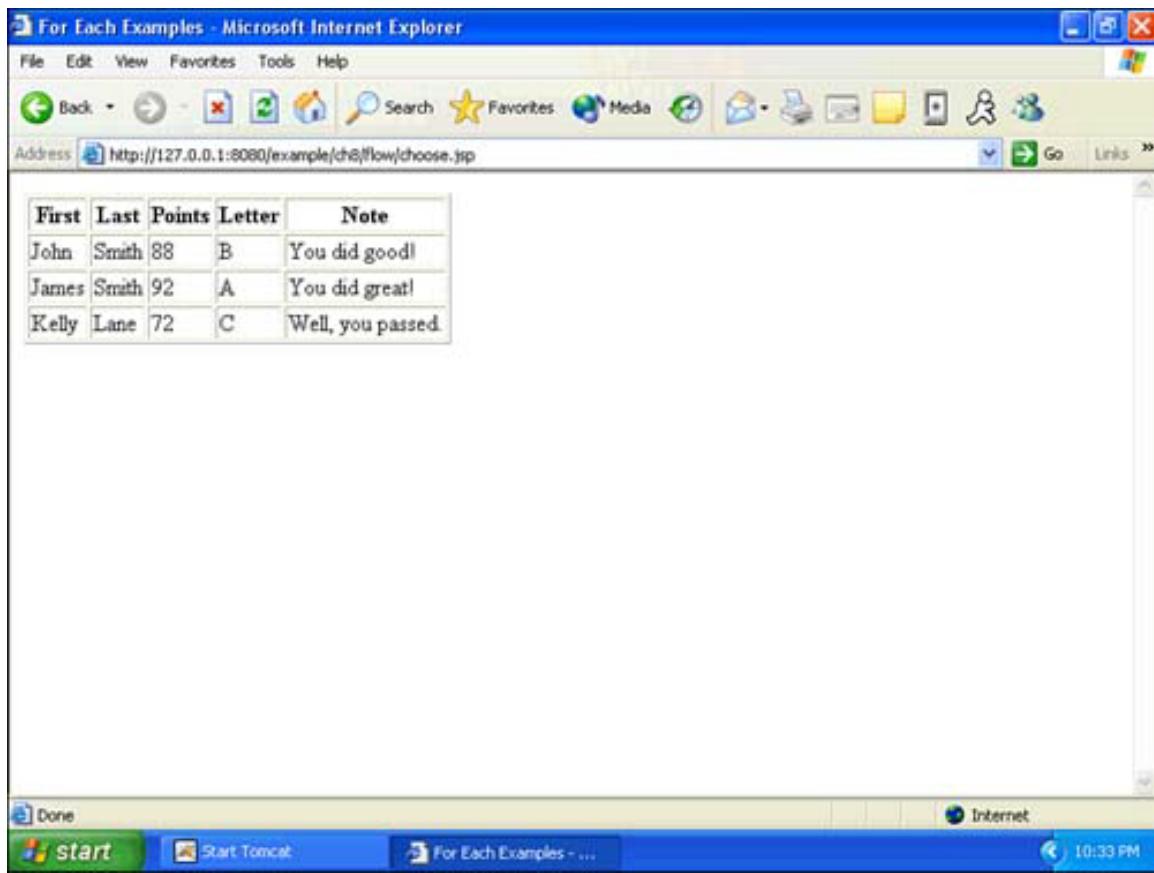
The `<x:choose>` tag works much like the `<c:choose>` tag. The `<x:choose>` tag allows you to nest several `<x:when>` and `<x:otherwise>` statements into a single comparison block. There is one form of the `<x:choose>` tag:

```
<x:choose>
body content (<x:when> and <x:otherwise> subtags)
</x:choose>
```

There are no attributes to this tag. The `<x:choose>` tag will begin evaluating the `<x:when>` statements until one matches. Our program executes the first matching `<x:when>` tag and then exits the `<x:choose>` block. It is impossible for more than one `<x:when>` block to execute. If no `<x:when>` tag executes, the program executes an `<x:otherwise>` tag if one exists.

Let's see an example of using the `<x:choose>` tag. This example extends our previous example by attaching a note to the score of each student. The program will display any one of several predefined messages. The message is determined by how well the user scored. We've shown the output from this program in [Figure 8.5](#).

Figure 8.5. Using the `<x:choose>` tag.



This program uses the same iteration loop we used in the previous example. Listing 8.6 shows the contents of this page.

Listing 8.6 The <x:choose> Tag (choose.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %><%@ taglib
    uri="http://java.sun.com/jstl/xml" prefix="x" %>
<html>
    <head>
        <title>For Each Examples</title>
    </head>

    <body>
        <c:import var="students" url="students.xml" />

        <x:parse var="doc" xml="${students}" />

        <table border="1">
            <tr>
                <th>First</th>
                <th>Last</th>
                <th>Points</th>
                <th>Letter</th>
                <th>Note</th>
            </tr>
```

```

<th>Note</th>
</tr>

<x:forEach var="student" select="$doc/students/student">
<tr>
    <td>
        <x:out select="name/first" />
    </td>

    <td>
        <x:out select="name/last" />
    </td>

    <td>
        <x:out select="grade/points" />
    </td>

    <td>
        <x:out select="grade/letter" />
    </td>

    <td>
        <x:choose>
            <x:when select="grade/points>90">You did
            great!</x:when>

            <x:when select="grade/points>80">You did
            good!</x:when>

            <x:when select="grade/points>75">You did
            ok.</x:when>

            <x:when select="grade/points>70">Well, you
            passed.</x:when>

            <x:otherwise>You failed</x:otherwise>
        </x:choose>
    </td>
</tr>
</x:forEach>
</table>
</body>
</html>

```

We use the `<x:choose>` tag to display the note for each student:

```
<x:choose>
```

The `<x:choose>` tag is similar to the `<c:choose>` tag. The primary difference is evident in the `<x:when>` tags. The `<x:when>` tags do not accept a test attribute; instead, they accept select attributes that specify an XPath expression that you want to evaluate. It is important that this XPath expression return a Boolean so that the `<x:when>` tags can take the most appropriate action.

The program begins by checking whether see if the XPath expression given by `grade/points>90` evaluates to true. If this expression does evaluate to true, then the program executes the body of the `<x:when>` tag and terminates the `<x:choose>` loop:

```
<x:when select="grade/points>90">You did  
great!</x:when>
```

If the first `<x:when>` tag does not execute, the page continues to examine `<x:when>` tags until one matches:

```
<x:when select="grade/points>80">You did  
good!</x:when>  
<x:when select="grade/points>75">You did  
ok.</x:when>  
<x:when select="grade/points>70">Well, you  
passed.</x:when>
```

If none of the `<x:when>` tags match, then our program executes the `<x:otherwise>` tag. If no `<x:otherwise>` tag exists, the `<x:choose>` tag will exit. The following `<x:otherwise>` tag causes the default message to appear:

```
<x:otherwise>You failed</x:otherwise>  
</x:choose>
```

Using the `<x:when>` Tag

The `<x:when>` tag is used in conjunction with the `<x:choose>` tag. The `<x:when>` tag checks to see whether the XPath expression evaluates to true; if it does, the program executes the body of the `<x:when>` tag. There is one form of the `<x:when>` tag:

```
<x:when select="XpathExpression">  
body content  
</x:when>
```

The `<x:when>` tag accepts one attribute:

Attribute	Required	Purpose
<code>select</code>	N	An XPath expression that you want to evaluate.

If the XPath expression evaluates to true, then the program executes the body of the `<x:when>` tag. If the XPath expression evaluates to false, then the next `<x:when>` tag is evaluated. The `<x:when>` tags are executed in the same order in which they are presented in the file. Only one `<x:when>` tag will execute per call to `<x:choose>`; the `<x:choose>` block exits after the first `<x:when>` tag matches.

Using the <x:otherwise> Tag

The `<x:otherwise>` tag works like a default `<x:when>` tag. If no other `<x:when>` tag evaluates to true in a `<x:choose>` block, then the `<x:otherwise>` tag will be executed. You should have only one `<x:otherwise>` tag per `<x:choose>` block. There is one form of the `<x:otherwise>` tag:

```
<x:otherwise>
conditional block
</x:otherwise>
```

No attributes are accepted by the `<x:otherwise>` tag. You have to embed the code that you want executed within the body of the `<x:otherwise>` tag.

Transforming XML with XSLT

Using the flow-control and core XML tags of JSTL, you can transform XML documents into nicely formatted HTML documents. JSTL supports another standard that makes it even easier to format XML. Using XSLT, you can specify the exact format in which you would like your XML document to be formatted.

A complete discussion of XSLT is beyond the scope of this book. If you are interested in learning XSLT, visit <http://www.w3.org/TR/xslt>. We also recommend *Sams Teach Yourself XSLT in 21 Days* (Sams, 2002, ISBN 0672323184).

We limit our discussion of XSLT to how XSLT is integrated with JSTL. JSTL allows you to specify two input files and receive the output. The input files are an XML data file and an XSLT formatting file. You specify the output in the XSLT formatting file. A properly written XSLT format file can output data in nearly any text format. The two most common forms are HTML and XML. XSLT is often used to generate HTML output from XML data. In addition, XSLT is used to translate between two incompatible XML formats.

The JSTL XML tags provide two tags that are used to work with XSLT files. The primary tag is the `<x:transform>` tag, which actually does the transformation. The second tag, `<x:param>`, allows you to specify parameters that will be passed to the XSLT script. These script parameters let you customize the operation of your XSLT script file.

[Listing 8.1](#) shows the XML file that we've used for all the examples in this chapter. Let's now see how to use XSLT with this sample file. To do this, we must use an XSLT file designed to work with the students.xml file. We've shown this XSLT file in [Listing 8.7](#).

Listing 8.7 A Sample XSLT File (transform.xsl)

```

<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="students">
        <html>
            <head>
                <title>XSLT Transform</title>
            </head>
            <body>
                <table border="1">
                    <tr><th>First</th><th>Last</th><th>Points</th><th>Letter</th>
                </tr>
                <xsl:apply-templates/>
            </table>
            </body>
        </html>
    </xsl:template>
    <xsl:template match="student">
        <tr>
            <td><xsl:value-of select="name/first"/></td>
            <td><xsl:value-of select="name/last"/></td>
            <td><xsl:value-of select="grade/points"/></td>
            <td><xsl:value-of select="grade/letter"/></td>
        </tr>
    </xsl:template>
</xsl:stylesheet>

```

Our XSLT file (in [Listing 8.7](#)) formats our XML file (shown in [Listing 8.1](#)) to look exactly like the HTML shown in [Figure 8.4](#).

Now, let's examine the basic structure of the XSLT file. This file is responsible for generating the entire HTML document—everything from the beginning `<html>` tag to the ending `</html>` tag. To do this, we use templates to handle specific blocks of the XML document. The node that encloses the entire student file is named `students`. An XSL template is provided that encapsulates this node, and because the `students` node is the root node, this will be the first template executed. The root template is shown here:

```
<xsl:template match="students">
```

As you can see, this template seeks a match on the value `students`. Since this is the root node, it will be the first template executed. Our template then goes on to set up the header information and begin the table:

```

<html>
<head>
<title>XSLT Transform</title>
</head>
<body>
    <table border="1">
        <tr><th>First</th><th>Last</th><th>Points</th><th>Letter</th>
    </tr>

```

Now that we have reached the body of the table, we can use the XSLT tag `<xsl:apply-templates>` to process the child nodes of the current tag. In this case, all student records will be processed:

```
<xsl:apply-templates/>
</table>
</body>
</html>
</xsl:template>
```

For each student record encountered, our program seeks a matching student `<xsl:template>:`

```
<xsl:template match="student">
<tr>
```

To display each of the attributes for the student, our program uses the `<xsl:value-of>` tag. We provide this tag with relative XPath expressions that specify the element we want to display. The following code displays the attributes for each student:

```
<td><xsl:value-of select="name/first"/></td>
<td><xsl:value-of select="name/last"/></td>
<td><xsl:value-of select="grade/points"/></td>
<td><xsl:value-of select="grade/letter"/></td>
</tr>
```

Now that you know how to construct a basic XSLT document, we'll show you how to use this document with the JSTL XML tags to create HTML. Let's begin by examining the `<x:transform>` tag.

Using the `<x:transform>` Tag

You use the `<x:transform>` tag to transform XML data based on an XSLT script. There are three forms of the `<x:transform>` tag:

```
// Syntax 1: No body content
<x:transform
  xml="XMLDocument" xslt="XSLTStylesheet"
  [xmlSystemId="XMLSystemId"] [xsltSystemId="XSLTSystemId"]
  [{var="varName" [scope="scopeName"] | result="resultObject"}]
// Syntax 2: Transformation parameters specified using body
<x:transform
  xml="XMLDocument" xslt="XSLTStylesheet"
  [xmlSystemId="XMLSystemId"] [xsltSystemId="XSLTSystemId"]
  [{var="varName" [scope="scopeName"] | result="resultObject"}]
  <x:param> actions
  </x:param>
</x:transform>
// Syntax 3: Using a body to specify both an XML document and
// transformation parameters
<x:transform
  xslt="XSLTStylesheet"
```

```

xmlSystemId="XMLSystemId" xsltSystemId="XSLTSystemId"
[ {var="varName" [scope="scopeName"] | result="resultObject"} ]
XML document being processed
optional <x:param> actions
</x:parse>

```

The first syntax format contains no body content; the XML document is specified using the `xml` attribute. The second syntax allows you to specify XSLT transformation parameters. The third syntax lets you specify both XSLT parameters and an XML document. The `<x:transform>` tag accepts the following attributes:

Attribute	Required	Purpose
<code>result</code>	N	A scoped variable of type <code>javax.xml.transform.Result</code> that specifies how the result should be processed.
<code>scope</code>	N	Specifies the scope of the scoped variable referenced by the <code>var</code> attribute.
<code>var</code>	N	Specifies a scoped variable that will hold the transformed document. If none is specified, the results will be written to the page.
<code>xml</code>	N	Specifies a string that holds the XML file to be used with the transformation.
<code>xslt</code>	N	Specifies a string that holds the XSLT file to be used with the transformation.
<code>xmlSystemId</code>	N	Specifies the URI of the XML document to be used.
<code>xsltSystemId</code>	N	Specifies the URI of the XSLT document to be used.

You must provide XML to the `<x:transform>` tag. The two syntaxes give you different ways to do this. In the first syntax, the XML data is provided through the attributes of the `<x:transform>` tag. The second syntax specifies the input XML through the body data contained in the `<x:transform>` tag.

The `<x:transform>` tag must be given a valid XML and XSLT document to work with. You can specify the XML document either by embedding it in the body of the tag or by referencing it in the `xml` attribute. You specify the XSLT document using the `xslt` attribute.

By default, the results of the transformation are written to the page. If a `var` attribute is specified, the contents of the transformation are stored to that variable. This variable is of type `org.w3c.dom.Document`. The attribute `scope` allows you to set the scope for the variable referenced by the `var` attribute.

You may also specify a `Result` object by using the `result` attribute. This value determines how the transformation will take place. The `Result` object is of type `javax.xml.transform.Result` and contains information that is needed to build a transformation result tree. A single property, `systemID`, is provided by the `Result` object.

This object contains the system identifier, or URI, of the transformation. For more information, refer to the Java API documentation for the `javax.xml.transform.Result` class.

You should also note that if the XSLT stylesheet is specified via the `xslt` attribute, the `<x:transform>` action may transparently cache transformer objects to improve performance. The `<x:transform>` tag is not required to fetch the contents of the `xslt` attribute each time the `<x:transform>` tag is used.

Let's look at an example of how we can use the `<x:transform>` tag in a JSP page. This example, shown in [Listing 8.8](#), is very short; nearly all the work is done in the XSLT file. The output from this program is exactly the same as for the `<x:forEach>` tag example shown in [Figure 8.4](#).

Listing 8.8 Using the <x:transform> Tag

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jstl/xml" prefix="x" %>
<c:import var="xml" url="students.xml" />

<c:import var="xslt" url="transform.xsl" />
<x:transform xml="${xml}" xslt="${xslt}" />
```

As you can see in [Listing 8.8](#), both the XML file and the XSL file are loaded into scoped variables. The `<x:transform>` tag is then used to format the XML using the XSL. The end result is that the student list is written to the page, as in [Figure 8.4](#).

This brings up an important point. We have seen that there are two ways you can format the XML file. The end result is the same; they both end up looking like [Figure 8.4](#). On one hand, we can use the `<x:forEach>` tag and use JSTL to display the XML. On the other hand, we can use XSLT to transform the XML into the same HTML output. Which method is better?

The answer is that it depends. There are no set rules. Often, it comes down to programmer preferences. However, you should keep a few things in mind. Formatting with XSL does not allow you to interact with other JSTL components, such as databases. If the XML to be displayed easily fits within an XSL template, then XSL is probably the better solution. However, if the display requires complex cross-referencing or other complex processing, using the other JSTL tags might be a better solution.

Using the <x:param> Tag

XSLT documents can accept parameters to modify their behavior. The `<x:param>` tag allows you to pass parameters to the XSLT to be used with the transformation. There is one form of the `<x:param>` tag:

```
// Syntax 1: Parameter value specified in attribute value
```

```

<x:param name="name" value="value"/>
// Syntax 2: Parameter value specified in the body content
<x:param name="name">
parameter value
</x:param>

```

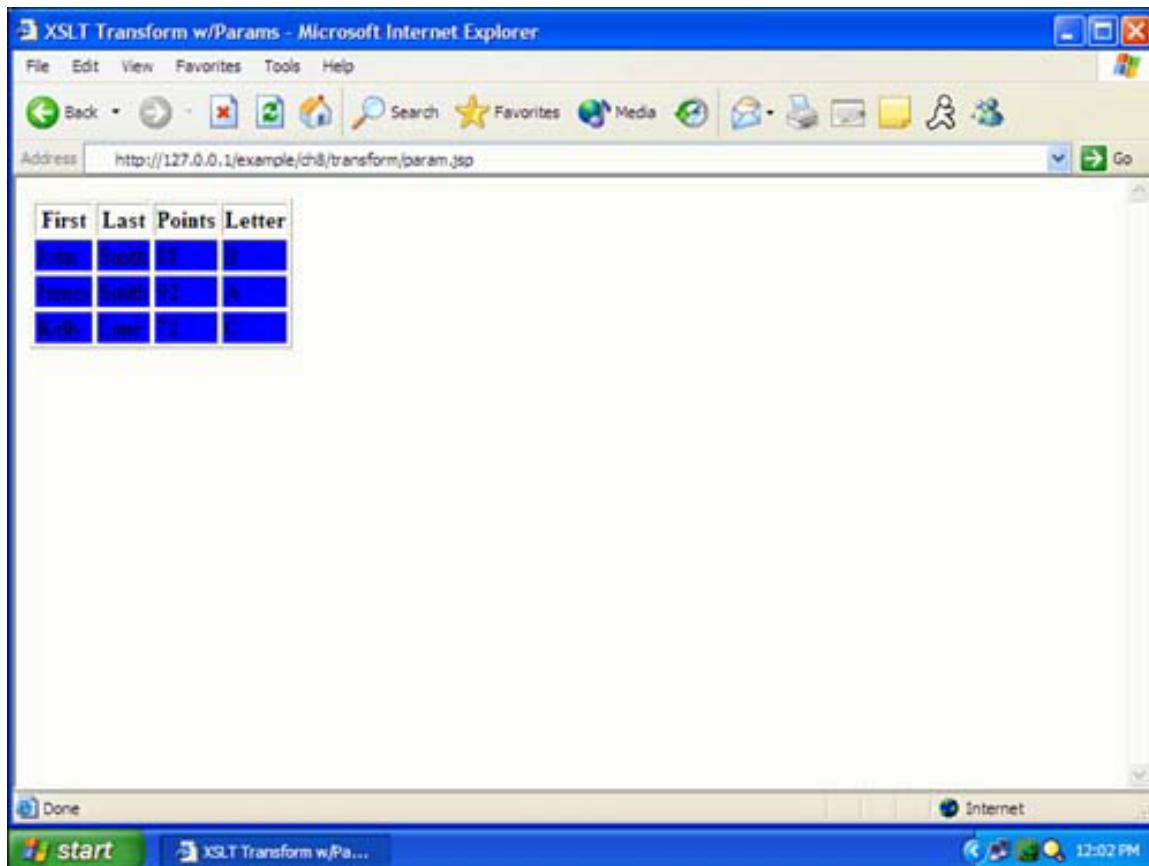
The attributes accepted by the `<x:transform>` tag are as follows:

Attribute	Required	Purpose
name	Y	Specifies the name of this parameter.
value	N	Specifies the value of this parameter.

The two syntaxes allow you to specify how the value of the parameter is to be specified. The first syntax uses an attribute to specify the value of the parameter. The second syntax uses the body of the `<x:param>` tag to specify the value of the parameter.

Let's look at an example of how to use XSLT parameters with JSTL. Our sample program displays the same student list. The only difference will be that this time we use an XSLT parameter to specify the background color of the table. The final output will look like [Figure 8.6](#).

Figure 8.6. Using parameters with XSLT.



This program requires a special version of the XSLT file. We have to modify the file to include the parameters, as shown in [Listing 8.9](#).

Listing 8.9 XSLT with Parameters (transformparam.xsl)

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:param name="bgColor"/>
    <xsl:template match="students">
        <html>
            <head>
                <title>XSLT Transform w/Params</title>
            </head>
            <body>
                <table border="1">
                    <tr><th>First</th><th>Last</th><th>Points</th><th>Letter</th>
                </tr>
                <xsl:apply-templates/>
            </table>
            </body>
        </html>
    </xsl:template>
    <xsl:template match="student">
        <tr>
            <td bgcolor="{$bgColor}"><xsl:value-of select="name/first"/>
        </td>
            <td bgcolor="{$bgColor}"><xsl:value-of select="name/last"/>
        </td>
            <td bgcolor="{$bgColor}"><xsl:value-of select="grade/points"/>
        </td>
            <td bgcolor="{$bgColor}"><xsl:value-of select="grade/letter"/>
        </td>
        </tr>
    </xsl:template>
</xsl:stylesheet>
```

The format of this XSLT file is similar to the one in [Listing 8.7](#). The primary difference is the built-in parameters. First, we have to declare the parameter. In this program, we use only one parameter, named bgColor:

```
<xsl:param name="bgColor"/>
```

Next, we declare the background color parameter. This parameter is used inside the <td> tags to set the background of the table:

```
<td bgcolor="{$bgColor}"><xsl:value-of select="name/first"/>
```

Finally, the background color must be specified by the calling JSTL. [Listing 8.10](#) shows our program.

Listing 8.10 Calling XSLT with Parameters

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jstl/xml" prefix="x" %>
<c:import var="xml" url="students.xml" />
<c:import var="xslt" url="transformparam.xsl" />
<x:transform xml="${xml}" xslt="${xslt}">
<x:param name="bgColor" value="blue"/>
</x:transform>
```

As you can see in Listing 8.10, we specified the color blue for the background color.

Summary

In this chapter, you learned how to access XML data with the XML tag library provided by JSTL. The JSTL XML tag library makes extensive use of XPath, a standard query language developed by the W3C that enables you to query XML documents. JSTL uses XPath to specify individual elements that you want to access, collections you want to iterate over, and expressions you want to use for Boolean comparisons.

Now you know how to process XML. As you learned in this chapter, XML is often obtained from files residing on the Web server. XML can also come from completely external sources. In the next chapter, we show you how to access external data sources. Often that data is XML, and can be processed using the tags we discussed in this chapter.

Chapter 9. Accessing Internet Resources with JSTL

In the previous chapter, you learned how to work with XML data, but we did not focus on where this data comes from. Much data is available on the Internet, in the form of XML and HTML.

The tags that we discuss in this chapter allow you to access information that is available on the Internet. Usually, this data is in XML, but it does not have to be. The JSTL tag library can easily access any form of text data stored on the Internet.

Accessing resources on the Internet requires the use of a URL. JSTL provides tags that let you easily format URLs for a variety of purposes. You can format URLs that include parameters for accessing external resources outside your Web page.

In addition to accessing external resources, you may want to access resources stored on your own Web site. You can use the URL tags to create URL strings that specify URLs within your Web site. This allows portions of your Web site to send parameters to other parts of your site. We begin this discussion by showing you the JSTL tags that enable you to access information on the Internet and manage URLs.

The URL-Related Tags

JSTL makes available tags that allow you to construct URLs, import data from external sources, and redirect users to new Web pages. The `<c:url>` and `<c:param>` tags enable you to construct URLs. Using the `<c:import>` tag, your program can access data from external sources, and using `<c:redirect>`, you can redirect your user to new Web pages. Let's examine all four of these tags, beginning with the `<c:import>` tag.

Using the `<c:import>` Tag

The `<c:import>` tag lets you import the contents of another Web page into your page. This data can be displayed directly to your page or it can be copied to a variable. There are two forms of the `<c:import>` tag:

```
//Syntax 1: Resource content inlined or exported as a String object
<c:import url="url" [context="context"]
[var="varName"] [scope="{page|request|session|application}"]
[charEncoding="charEncoding"]>
optional body content for <c:param> subtags
</c:import>
//Syntax 2: Resource content exported as a Reader object
<c:import url="url" [context="context"]
varReader="varReaderName"
```

```
[charEncoding="charEncoding"]>
body content where varReader is consumed by another action
</c:import>
```

The `<c:import>` tag accepts these attributes:

Attribute	Required	Purpose
<code>charEncoding</code>	N	Allows you to specify the character encoding (for example, UTF16).
<code>context</code>	N	Specifies the base URL that will be used to resolve a relative URL given by the <code>url</code> attribute.
<code>scope</code>	N	Specifies the scope for the scoped variable referenced by the <code>var</code> attribute. The default is page.
<code>url</code>	Y	Specifies the URL from which the import is to take place.
<code>var</code>	N	Specifies a variable that will receive the output from the specified URL.

In the first syntax, the content of the Internet resource is written to the page—unless a `var` attribute has been specified. In that case, the content of the Internet resource is written to the scoped variable specified by the `var` attribute.

The second syntax stores the content of the Internet object as a `java.io.Reader` object. This Reader object does have an important restriction: It cannot be accessed beyond the closing `<c:/import>` tag.

It is also important to remember that if you're using syntax 2, you cannot nest `<c:param>` tags. This is because these tags are not able to change the URL that has been requested. To address this problem, you should construct a URL using the `<c:url>` tag and use that URL as input to the `<c:import>` tag. To use the `<c:import>` tag to simply output to the browser, type the following command:

```
<c:import url="http://www.yahoo.com"/>
```

To copy the contents of a Web site to a variable, use this:

```
<c:import url="http://www.yahoo.com" var="contents"/>
```

This code copies the contents of <http://www.yahoo.com/> into a variable named `contents`.

It is also important to understand how character encoding and relative URLs are handled by the `<c:import>` tag. In the next two sections, we describe character encoding and relative URLs in greater detail.

Character Encoding

If no `charEncoding` attribute is specified, the following rules apply:

- If the response has content information (for example, `URLConnection.getContentEncoding()` has a non-null value), then the specified character encoding is used.
- Otherwise, ISO-8859-1—the default content type of a JSP page—is used.

NOTE

The `charEncoding` attribute should normally be required only when you're accessing non-HTTP resources because HTTP resources generally specify their correct encoding.

WARNING

Not all character encodings are supported by JSTL. One very common encoding that is not supported is GZIP. GZIP allows Web pages to compress their HTML files by applying the GZIP compression algorithm. If you attempt to access a Web page that uses GZIP encoding, an invalid encoding exception will be thrown. Short of implementing your own `<c:import>` tag, there is no workaround. This problem may not exist in all implementations or versions of JSTL.

Handling Relative and Absolute URLs

Let's see how the `<c:import>` tag handles relative and absolute URLs. We'll start with a relative URL from the same context.

Relative URL—Same Context

In this section, we'll show you what is meant by a relative URL that has the same context as the page using the URL. Consider two Internet resources that belong to the same Web application. When a relative URL occurs within the same context as the page using it, that URL is processed in the same way as the `<jsp:include>` include tag of the JSP specification.

According to the JSP specification, there are two types of relative URL. Relative URLs can either be *context-relative* paths or *page-relative* paths. A context-relative path is a path that begins with a /. A URL such as this is to be interpreted as relative to the application to which the JSP page belongs. A page-relative path is a path that does not start with a / and that is interpreted as relative to the current JSP page, as defined by the rules of inclusion of the `<jsp:include>` action in the JSP specification.

Keep in mind that the context for a relative URL is the Web Archive (WAR) file. If multiple Web applications are run from the same server, they will not share application- or session-scoped variables, even if they can be referenced using the same base as a relative URL.

Relative URL—Foreign Context

If the resource being imported belongs to a different Web application, then its context is considered foreign. In this case, you have to specify the context name via the context attribute.

The relative URL must be context-relative and start with a / since the including page does not belong to the same context. Likewise, the context name must also start with a /.

Absolute URLs

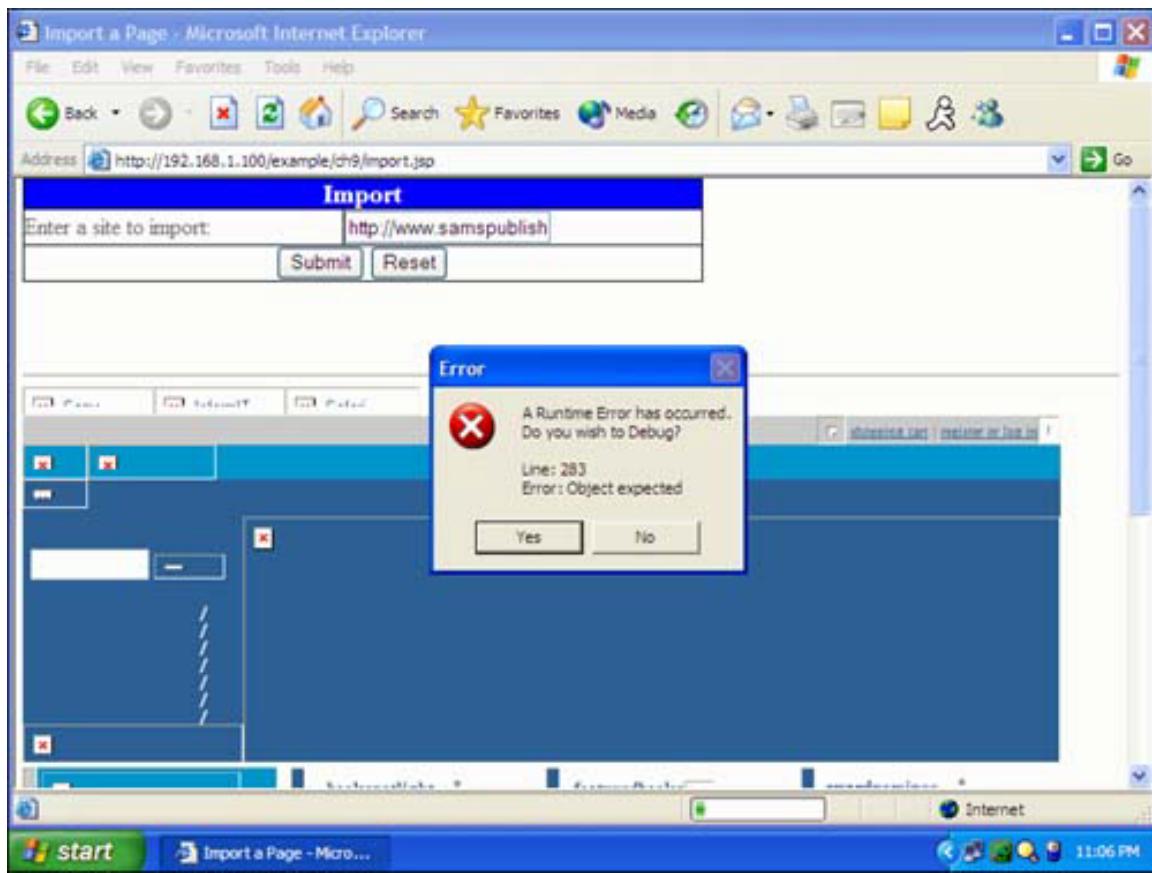
Absolute URLs are those stored in the java.net.URL and java.netURLConnection classes. Because JSP 1.2 (upon which JSTL is built) requires a J2SE platform at a minimum, the `<c:import>` action supports at a minimum the protocols offered in that platform for absolute URLs. More protocols can still be available to a Web application, but this will depend on the class libraries made available to the Webapp by the platform the container runs on. An example of this is the HTTPS protocol, which is not supported by most older JDKs yet could be added.

Let's look at a simple example that uses the `<c:import>` tag. (We'll examine a more complex example toward the end of this chapter.) Our first example asks the user to enter an HTTP address into a simple form. Once the user enters this address, the program will attempt to import the data of that Web page into the current one. The result is a page that displays the form near the top and the contents of the entered Web site at the bottom.

Only the HTML of the target Web site is retrieved. As a result, many of the images won't display. Further, if the site uses external JavaScript code that is accessed from a different page, these features won't be available either.

Generally, a program will not directly access HTML data from Web sites; the JSP page will access XML data that will be reprocessed and displayed, as we saw in [Chapter 8](#), "Accessing Data with XML Tags." However, for this example we simply repeat the HTML data located at the specified URL. You can see this program running in [Figure 9.1](#).

Figure 9.1. Displaying `http://www.samspublishing.com/`.



As you can see in Figure 9.1, the Web page is being instructed to import and then display the contents of the Sams Publishing Web page at <http://www.sampublishing.com/>. The images are not displayed, and the user sees a JavaScript error message, indicating that external JavaScript code cannot be accessed. However, despite these shortcomings you can clearly see that the `<c:import>` tag has successfully imported the HTML of the Sams Publishing site. This program is shown in Listing 9.1.

Listing 9.1 Importing Data (import.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt" %>
<html>
    <head>
        <title>Import a Page</title>
    </head>

    <body>
        <form method="POST">
            <table border="1" cellpadding="0" cellspacing="0"
                style="border-collapse: collapse" bordercolor="#111111"
                width="62%" id="AutoNumber1">
                <tr>
                    <td width="100%" colspan="2" bgcolor="#0000FF">
                        <p align="center">
                            <b>
```

```

        <font color="#FFFFFF" size="4">Import
        </font>
    </b>
</p>
</td>
</tr>

<tr>
    <td width="47%">Enter a site to import:</td>

    <td width="53%">
        <input type="text" name="url" size="20"
               value="http://www.samspublishing.com"/>
    </td>
</tr>

<tr>
    <td width="100%" colspan="2">
        <p align="center">
            <input type="submit" value="Submit" name="submit" />

            <input type="reset" value="Reset" name="reset" />
        </p>
    </td>
</tr>
</table>

<p>&#160;</p>
</form>

<c:if test="${pageContext.request.method=='POST' }">
    <hr>
    <c:import url ="${param.url}"/>
</c:if>
</body>
</html>

```

Like many of the examples in this book, Listing 9.1 displays a form from a JSP page and instructs the JSP page to post to itself. A `<c:if>` tag near the bottom of the sample program shows where the page checks to see whether it is being posted to:

```

<c:if test="${pageContext.request.method=='POST' }">
    <hr>
    <c:import url ="${param.url}"/>
</c:if>

```

The `<c:import>` tag we used here is quite simple. The only attribute we specified is the `url` attribute, which indicates which URL we want to use to retrieve the Internet resource. The contents of this page will be written right to the page since no `var` attribute was specified.

Using the <c:url> Tag

The <c:url> tag is used to format a URL. Using this tag, you can easily construct URLs that resolve relative paths and parameters. There are two forms of the <c:url> tag:

```
// Syntax 1: Without body content
<c:url value="value" [context="context"]
[var="varName"] [scope="{page|request|session|application}"]/>
// Syntax 2: With body content to specify query string parameters
<c:url value="value" [context="context"]
[var="varName"] [scope="{page|request|session|application}"]>
<c:param> subtags
</c:url>
```

The attributes accepted by the <c:url> tag are as follows:

Attribute	Required	Purpose
context	N	Specifies the base URL that will be used to resolve a relative URL given by the value attribute.
scope	N	Specifies the scope for the scoped variable referenced by the var attribute. The default is page.
value	Y	Specifies the URL that the import is to take place from.
var	N	Specifies a variable that will receive the output from the specified URL.

In the first syntax, the entire URL is specified inline. For simple parameterless URLs, you would probably use the first syntax.

The second syntax lets you specify <c:param> tags. These tags will be discussed in the next section.

The URL will be written to the scoped variable referenced by the var attribute. The base URL that is to be processed should be stored in the value tag. The scope attribute determines what scope the scoped variable referenced by the var tag should attain.

We will not look at an example of the <c:url> tag by itself. The <c:url> tag is almost always used in conjunction with the <c:param> tag. In the next section, we examine the <c:param> tag.

Using the <c:param> Tag

The <c:param> tag can be attached to either the <c:import> tag or the <c:url> tag to specify URL parameters. These parameters are of type

<http://www.jeffheaton.com/cgi-bin/test.cgi?parama=test¶mb=testing>.

This URL would specify two parameters, named parama and paramb, that would hold the values test and testing. There are two forms of the <c:param> tag:

```

//Syntax 1: Parameter value specified in attribute value
<c:param name="name" value="value"/>
//Syntax 2: Parameter value specified in the body content
<c:param name="name">
parameter value
</c:param>

```

The `<c:param>` tag accepts these attributes:

Attribute	Required	Purpose
<code>name</code>	Y	Specifies the name of this parameter.
<code>value</code>	N	Specifies the value of this parameter.

The first syntax produces a URL using the `value` attribute to specify the value of the tag. This works well for short values. If the value that you would like to assign is long, you should use the second version of the `<c:param>` tag, which uses the body of the tag to specify its value. If you are using the second syntax, you must not specify a `value` attribute.

One important reason to use the `<c:param>` tag involves URL encoding. URL encoding is the process by which characters that are not allowed on a URL line are encoded properly. For example, the parameter Hello World would be encoded to Hello%20World, since the space is not a valid character for a URL.

Let's now examine a sample program that use the `<c:import>` and `<c:param>` tags. This program constructs a parameterized URL based on the specified base URL and parameters. Figure 9.2 shows our program, and Listing 9.2 shows how it was constructed.

Listing 9.2 Constructing URLs (url.jsp)

```

<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt" %>
<html>
    <head>
        <title>Create a URL</title>
    </head>

    <body>
        <form method="POST">
            <table border="1" cellpadding="0" cellspacing="0"
                style="border-collapse: collapse" bordercolor="#111111"
                width="62%" id="AutoNumber1">
                <tr>
                    <td width="100%" colspan="2" bgcolor="#0000FF">
                        <p align="center">
                            <b>
                                <font color="#FFFFFF" size="4">URL
                                </font>
                            </b>
                        </p>
                    </td>
                </tr>
            </table>
        </form>
    </body>
</html>

```

```

        </td>
    </tr>

    <tr>
        <td width="47%">Enter a base URL:</td>
        <td width="53%">
            <input type="text" name="url" size="20"
                   value="http://www.samspublishing.com"/>
        </td>
    </tr>

    <tr>
        <td width="47%">Enter a value for parameter "parm1"</td>
        <td width="53%">
            <input type="text" name="parm1" size="20" />
        </td>
    </tr>

    <tr>
        <td width="47%">Enter a value for parameter "parm2"</td>
        <td width="53%">
            <input type="text" name="parm2" size="20"/>
        </td>
    </tr>

    <tr>
        <td width="47%">Enter a value for parameter "parm3"</td>
        <td width="53%">
            <input type="text" name="parm3" size="20"/>
        </td>
    </tr>

    <tr>
        <td width="100%" colspan="2">
            <p align="center">
                <input type="submit" value="Submit" name="submit" />
                <input type="reset" value="Reset" name="reset" />
            </p>
        </td>
    </tr>
</table>

<p>&#160;</p>
</form>

<c:if test="${pageContext.request.method=='POST'}">
    <hr>
    <c:url value="${param.url}" var="url">
        <c:param name="parm1" value="${param.parm1}" />
        <c:param name="parm2" value="${param.parm2}" />
        <c:param name="parm3" value="${param.parm3}" />
    </c:url>

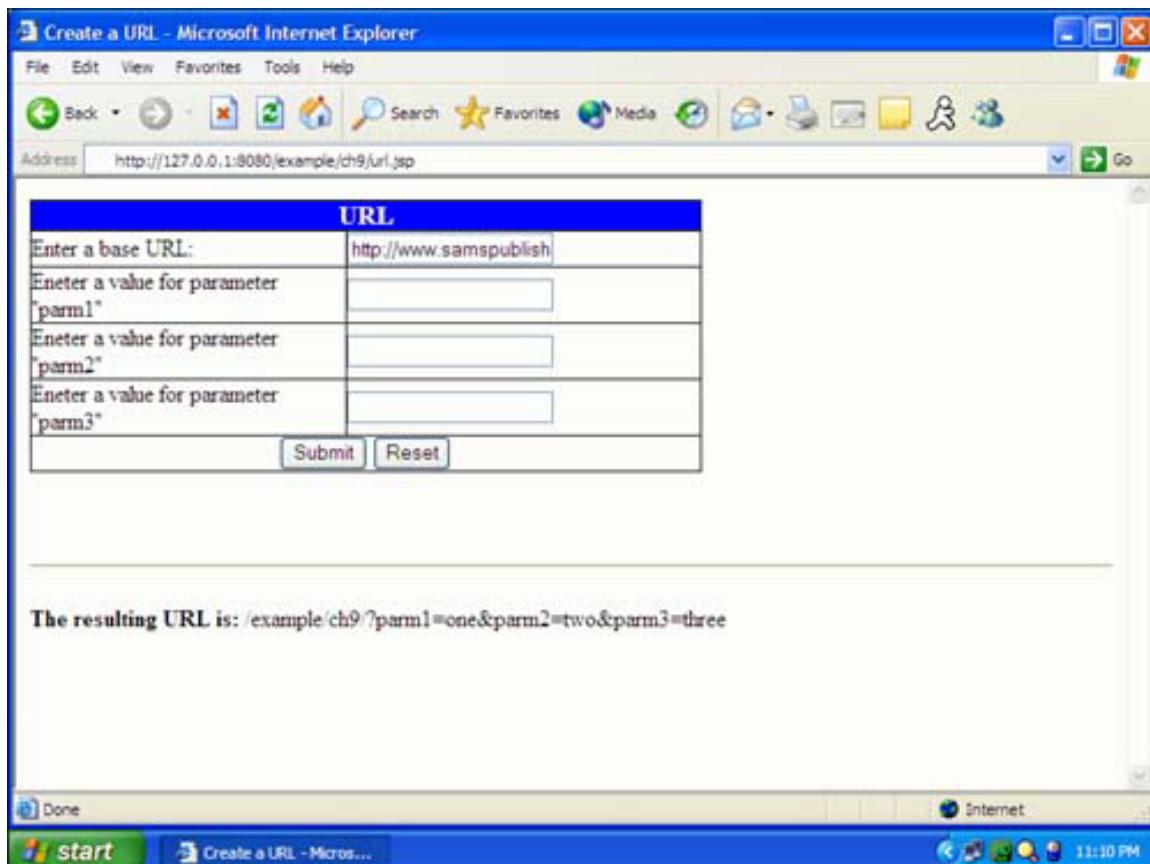
```

```

<br/><b>The resulting URL is:</b>
<c:out value="${url}" />
</c:if>
</body>
</html>

```

Figure 9.2. Constructing URLs.



The program in Listing 9.2 works like many other examples in this book in that it displays a form and posts back to itself. After the page displays, the form checks to see whether any data was posted to it:

```
<c:if test="${pageContext.request.method=='POST'}">
```

If data was posted to the page, then we read the base URL and three parameters from the form data. This data is used to construct a URL. The following code reads in the parameters and constructs the URL:

```

<hr>
<c:url value="${param.url}" var="url">
  <c:param name="parm1" value="${param.parm1}" />
  <c:param name="parm2" value="${param.parm2}" />
  <c:param name="parm3" value="${param.parm3}" />
</c:url>

```

As you can see, this code implements a single `<c:url>` tag that has three nested `<c:param>` tags. This allows a URL to be constructed that has three parameters. All of the data that is to be formed into the URL comes from the param collection, which contains the data that was posted from the form:

```
<br/><b>The resulting URL is:</b>
<c:out value="${url}" />
</c:if>
```

Using the `<c:redirect>` Tag

It is often useful to redirect the user from one Web page to another. An example of this might be a login page to a Web site. The page displays a form that asks users to enter their login information; the form posts to the same JSP page that displays it. The JSP page validates the user, and if a successful login is achieved, the user is redirected to the main page of the secure area of the Web site. The `<c:redirect>` tag has two forms:

```
// Syntax 1: Without body content
<c:redirect url="value" [context="context"] />
// Syntax 2: With body content to specify query string parameters
<c:redirect url="value" [context="context"] />
<c:param> subtags
</c:redirect>
```

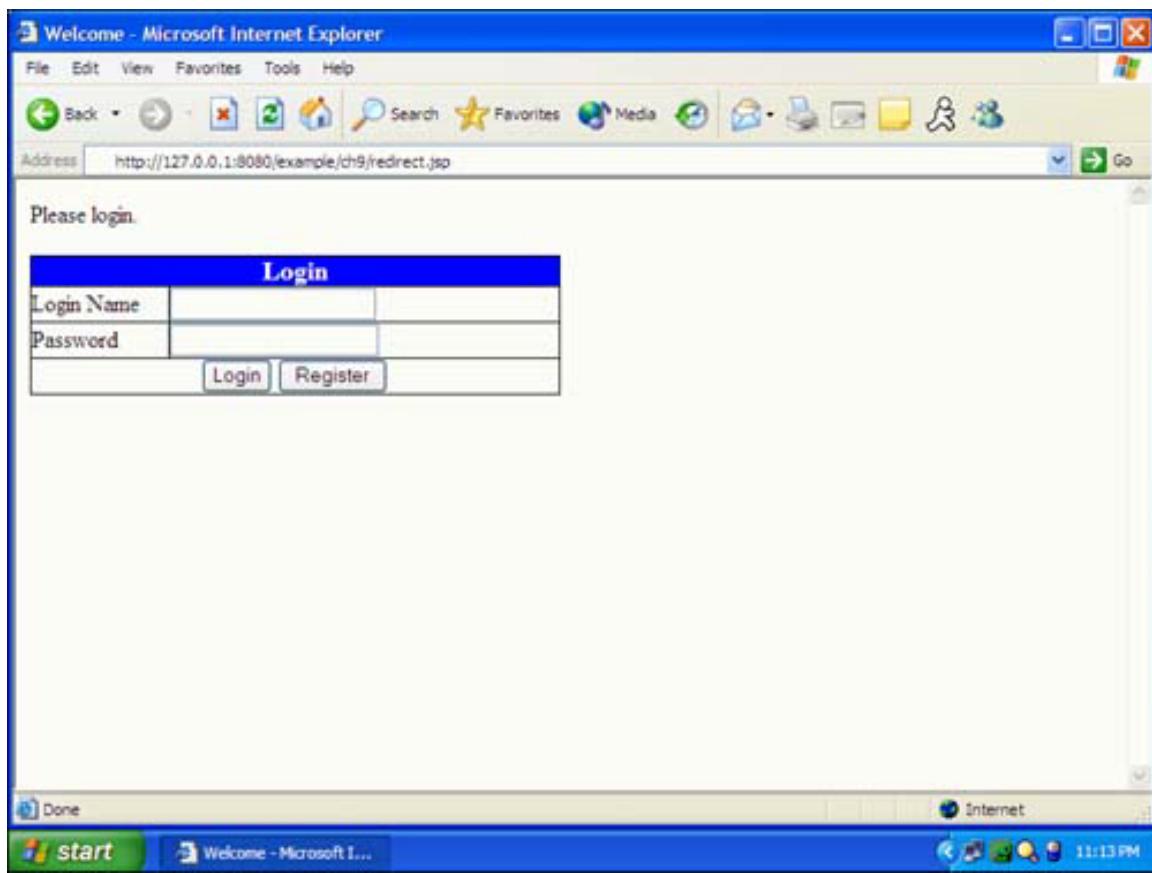
The attributes accepted by the `<c:redirect>` tag are as follows:

Attribute	Required	Purpose
<code>context</code>	N	Specifies the base URL that will be used to resolve a relative URL given by the <code>url</code> attribute.
<code>url</code>	Y	Specifies the URL that the user is taken to.

The `<c:redirect>` tag works much like the `<c:url>` tag in that it can accept parameters in the form of `<c:param>` tags. You append these parameters to the URL in the same way you do for the `<c:url>` tag. Keep in mind that once a `<c:redirect>` tag completes, the current page stops execution. A successful `<c:redirect>` will always be the last line executed for the current page because it transfers control to the redirected page.

Let's look at an example that uses the `<c:redirect>` tag. This program presents a simple login screen (similar to our forum example from [Chapter 7](#), "Accessing Data with SQL Tags"). This login form allows the user to enter an ID and a password. If the user is not already registered with the system, the user has the option of clicking the Register button. Clicking this button takes users to a screen that allows them to create a user account. Getting to this registration screen is accomplished with the `<c:redirect>` tag. Our login program is shown in [Figure 9.3](#).

Figure 9.3. Using the `<c:redirect>` tag.



When users click the Register button, they are redirected to a file named register.html. Let's take a look at [Listing 9.3](#) to see how this redirection fits into the logical flow of the login process.

Listing 9.3 Redirecting (redirect.jsp)

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

<c:if test="${pageContext.request.method=='POST'}">
  <c:if test="${param.reg!=null}">
    <c:redirect url="register.html" />
  </c:if>

  <h3 color="red">Sorry, we have no one registered with that
  name.</h3>
</c:if>

<html>
  <head>
    <title>Welcome</title>
  </head>

  <body>
```

```

Please login.<br>
<form method="POST">
    <table border="1" cellpadding="0" cellspacing="0"
        style="border-collapse: collapse" bordercolor="#111111"
        width="49%" id="AutoNumber1">
        <tr>
            <td width="100%" colspan="2" bgcolor="#0000FF">
                <p align="center">
                    <b>
                        <font color="#FFFFFF" size="4">Login</font>
                    </b>
                </p>
            </td>
        </tr>

        <tr>
            <td width="26%">Login Name</td>
            <td width="74%">
                <input type="text" name="uid" size="20" />
            </td>
        </tr>

        <tr>
            <td width="26%">Password</td>
            <td width="74%">
                <input type="password" name="pwd" size="20" />
            </td>
        </tr>

        <tr>
            <td width="100%" colspan="2">
                <p align="center">
                    <input type="submit" value="Login" name="Login" />
                    <input type="submit" value="Register" name="reg" />
                </p>
            </td>
        </tr>
    </table>

    <p>&#160;</p>
</form>

</body>
</html>

```

The program in Listing 9.3 begins by displaying an ordinary login form. The user ID is stored in the parameter uid, and the password is stored in the parameter pwd. When the user clicks either of the input buttons, this form posts back to the same page, which is `redirect.jsp`. Once this page posts back to itself, the program executes the following JSTL code:

```
<c:if test="${pageContext.request.method=='POST'}">
```

Once the program has determined that data has been posted to the page, the following code is executed:

```
<c:if test="${param.reg!=null}">
```

First, we must determine whether the user clicked the Register button. If a button is clicked, a parameter is registered that has the same name as the button. So, to determine whether the user clicked the Register button, we simply check to see whether the param.reg variable is equal to null. If this value is not null, then we know that the Register button was clicked:

```
<c:redirect url="register.html" />
```

Now that we have determined that the user clicked the Register button, we can redirect the user to a page that can handle the registration. Our `<c:redirect>` tag transfers the user to the register.html page. This is the page that the user is viewing. The browser also reflects this in the URL line by indicating that the user is now viewing the register.html page:

```
</c:if>
```

It is important to note that the `<c:redirect>` tag causes execution of the current JSP page to stop immediately. Any statements that follow the `<c:redirect>` tag will never be executed. This includes the following lines of JSTL code:

```
<h3 color="red">Sorry, we have no one registered with that  
name.</h3>  
</c:if>
```

This code displays a message that the user ID is invalid. Since this example serves only to show how the redirect works, we did not implement the actual login logic, as we did in [Chapter 7](#). This program simply shows how to redirect to a new page depending on which button the user clicks.

The preceding sections have shown you how to use the URL-related tags of JSTL. We have seen some examples that carry out basic URL- and Internet-related functions. Now, let's look at an example that shows how to access real, publicly available data that is stored on the Internet. To do this, we must discuss the sources of information available for access by the URL-related tags.

Understanding Internet Resources

Numerous sources of XML information are available on the Internet. Many are subscription-based and require you to make payments to their provider. There are also many free XML resources that you can take advantage of on the Internet.

One benefit of using Internet resources is that many times they allow two platforms to work together. Microsoft Passport is an example of Java working with a subscription-based XML service available on the Internet. Passport attempts to solve the problem of identifying users. Often, a user will maintain accounts with several different subscription-based Web sites that the user frequents. This requires the user to register with each individual Web site.

Microsoft Passport attempts to alleviate this burden by having the user register just once, and then provides a universal registration to every site that the user frequents. This universal login is provided by the Microsoft Passport Service.

In order for a typical Web site to maintain such a relationship with Microsoft Passport, the site must exchange information with Microsoft. To accomplish this, XML is used. This is an example of how a Web site might use the Internet resource tags provided by JSTL.

There are also many free sources of XML data that you can integrate into your application. One such service is RSS, which we use for many of this chapter's examples. Let's take a look.

Introducing RSS

Rich Site Summary (RSS) is an XML standard for making the headlines available through a news Web site available to the other Web sites. This standard was introduced by Netscape. You can obtain the official specification from <http://my.netscape.com/publish/formats/rss-spec-0.91.html>.

This relationship is good for both the site that produces the news stories as well as the site that displays the stories. Sites that receive the stories are given a source of fresh content that can easily be integrated right into their Web site. For news producers, RSS can increase traffic, since all exported stories contain links back to the story at their site.

Many news-related sites make RSS feeds available. Table 9.1 lists some of the commonly used RSS feeds.

Table 9.1. Some RSS Feeds

Web Site	RSS Feed URL
FreshMeat	http://freshmeat.net/backend/fm.rdf
MacWeek	http://macweek.zdnet.com/macweek.xml
Motley Fool	http://www.fool.com/About/headlines/rss_headlines.asp
NewsForge	http://www.newsforge.com/newsforge.rdf

NewsForge	http://www.newsforge.com/newsforge.rss
Reuters Health	http://www.reutershealth.com/eline.rss
Salon	http://www.salon.com/feed/RDF/salon_use.rdf
Slashdot	http://www.slashdot.org/slashdot.rdf
Wired	http://www.wired.com/news_drop/netcenter/netcenter.rdf

The RSS format is a relatively simple XML format. Listing 9.4 shows a sample of RSS data.

Listing 9.4 A Sample RSS Feed File (sample.rss)

```

<?xml version="1.0" ?>
<!DOCTYPE rss PUBLIC "-//Netscape Communications//DTD RSS 0.91//EN"
 "http://www.wired.com/news_drop/rss-0_91.dtd">
<rss version="0.91">
  <channel>
    <title>My Information Service</title>
    <link>http://www.jeffheaton.com/</link>
    <description>
      This is where the description for this channel goes.
    </description>
    <language>en-us</language>
    <pubDate>Tue, 21 May 2002 17:38 PDT</pubDate>
    <image>
      <title>My Image Title</title>
      <url>http://www.jeffheaton.com/images/jeffpic1.jpg</url>
      <link>http://www.jeffheaton.com</link>
    </image>

    <item>
      <title>First Story Title</title>
      <link>http://www.jeffheaton.com/news/story1.html</link>
      <description>
        This is where the brief description to the first story
        belongs.
      </description>
    </item>

    <item>
      <title>Second Story Title</title>
      <link>http://www.jeffheaton.com/news/story2.html</link>
      <description>
        This is where the brief description to the second story
        belongs.
      </description>
    </item>

    <item>
      <title>Third Story Title</title>
      <link>http://www.jeffheaton.com/news/story3.html</link>
      <description>
        This is where the brief description to the third story
      </description>
    </item>
  </channel>
</rss>
```

```
    belongs.  
    </description>  
  </item>  
  
</channel>  
</rss>
```

Let's examine the major tags used by the XML-based RSS file format. As you can see in Listing 9.4, the entire file is contained in the `<rss>` tag. This tag can contain one or more channels. Most RSS feeds will have a single channel.

The `<channel>` tag encloses one news feed. Inside the `<channel>` tag, you first encounter some tags that provide general information about the channel. The first piece of information is usually the title:

```
<title>My Information Service</title>
```

The title is typically displayed on a Web site with a link back to the original news site that provided the story. The Web application must know where this link should point. The destination of the title `<link>` tag should specify the Web address that the news provider would like people to visit when they click on the title. The following is an example of a `<link>` tag:

```
<link>http://www.jeffheaton.com/</link>
```

In addition to the link and title, the RSS format specifies that a description of the channel be provided. The following code shows the `<description>` tag:

```
<description>  
  This is where the description for this channel goes.  
</description>
```

The RSS standard also allows for languages other than English to be used within RSS feeds. To do this, you must specify the language that your site is in by using the `<language>` tag. The following `<language>` tag specifies that U.S. English is the language being used:

```
<language>en-us</language>
```

How current news stories are often a critical factor in their success. The `<pubDate>` tag allows you to specify the date that the channel was published. Here's an example of the `<pubDate>` tag:

```
<pubDate>Tue, 21 May 2002 17:38 PDT</pubDate>
```

In addition to the title provided by the RSS feed file, you are given an image to display. The image is determined by the tags beneath the `<image>` tag. The following shows how the `<image>` tag is represented in an RSS feed file:

```

<image>
  <title>My Image Title</title>
  <url>http://www.jeffheaton.com/images/jeffpic1.jpg</url>
  <link>http://www.jeffheaton.com</link>
</image>

```

As you can see, the `<image>` node contains a number of subnodes that describe the image. The `<title>` tag contains the title of the image, which is usually specified as a part of the HTML `` tag's ALT attribute. The `<url>` tag specifies the URL of the image, which is included in the display. Finally, the `<link>` tag specifies the URL that users will be taken to if they click the image.

These tags provide the header information for an RSS news feed. Now, let's look at the individual news stories contained in a feed. A news story is stored in an `<item>` tag:

```
<item>
```

Each story is provided with a title. The title is specified using the `<title>` tag:

```
<title>First Story Title</title>
```

Usually, this title will be linked to the actual story on the news provider's Web site. The following `<link>` tag specifies the URL to be accessed if the user clicks the title:

```
<link>http://www.jeffheaton.com/news/story1.html</link>
```

A short description accompanies the RSS news story. This allows readers to get a general idea of what the story is about before they click the link that will take them back to the news publisher's site:

```

<description>
  This is where the brief description to the first story belongs.
</description>
</item>

```

Now that you understand the basic format of the RSS XML file format, let's look at a simple Web application that displays the headlines of the news provider.

Using the RSS Format

When executed, our sample Web page will display a table that tells readers what news stories are available from the Wired RSS feed. This program is shown in [Listing 9.5](#).

Listing 9.5 Reading from an RSS feed(news.jsp)

```

<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jstl/xml" prefix="x" %>
<html>
  <head>

```

```

<title>Reading RSS</title>
</head>

<body>
<c:import var="news"
url="http://www.wired.com/news_drop/netcenter/netcenter.rdf" />

<x:parse var="doc" xml="${news}" />

<table border="1" width="100%">
<tr bgcolor="blue">
<td align="center">
<font color="white" size="+2">
<b>
<x:out select="$doc/rss/channel/title" />
</b>
</font>
<br />

<font color="white" size="-2">
<x:out select="$doc/rss/channel/pubDate" />
</font>
</td>
</tr>
<tr>
<td valign="top">
<x:out select="$doc/rss/channel/description" />
</td>
</tr>

<x:forEach var="story" select="$doc/rss/channel/item">
<tr bgcolor="blue">
<td align="center">
<a href="">
<font color="white">
<x:out select="title" />
</font>
</a>
</td>
</tr>
<tr>
<td valign="top">
<x:out select="description" />
</td>
</tr>
</x:forEach>
</table>
</body>
</html>

```

The first thing that our program does is connect to the Wired RSS feed:

```
<c:import var="news"
```

```
url="http://www.wired.com/news_drop/netcenter/netcenter.rdf" />
```

As you can see, the contents of the Wired RSS feed are copied into the scoped variable news. This XML data must then be parsed:

```
<x:parse var="doc" xml="${news}" />
```

With the document properly parsed, our program is ready to display the header information. The program displays this information in an HTML table.

The title of the news story is displayed using an `<x:out>` tag, and can easily be accessed using an XPath. The following tag displays the title:

```
<x:out select="$doc/rss/channel/title" />
```

The publication date and description are displayed in a similar manner:

```
<x:out select="$doc/rss/channel/pubDate" />
<x:out select="$doc/rss/channel/description" />
```

The individual stories are contained in a collection. To display the stories, we use the `<x:forEach>` tag:

```
<x:forEach var="story" select="$doc/rss/channel/item">
```

Individual stories can now be displayed as the `<x:forEach>` tag iterates through each story in the collection:

```
<a href="">
<font color="white">
  <x:out select="title" />
```

Our program displays the title with an enclosing `<a>` tag that provides the hyperlink to the real story. The description of the story is displayed using the `<x:out>` tag:

```
<x:out select="description" />
```

As you can see, the RSS format can easily be integrated into most Web sites. Other XML formats are available that you can include just as easily into your Web site, and still others are being created. By using XML from external resources, you can keep your Web site content fresh. The URL tags allow you to take advantage of XML data sources.

Summary

In this chapter, we explained how to use the URL-related tags provided by JSTL. We can use these tags to construct URLs that include parameters that enable communication to

take place across Web pages. In the next chapter, we introduce I18N programming in JSTL.

Chapter 10. Understanding JSTL Internationalization

Java was designed to support internationalization. All strings in Java are stored internally as Unicode strings, which normally are transparent to the programmer. However, when your program has to support a language such as Japanese or Chinese, you must use Unicode.

One of the most common reasons for making a program international is the need to display the program's data in a variety of languages. JSTL provides several tags that make it easy to create a multilingual program. In this chapter, we show you how to make our forum example from [Chapter 7](#) work with multiple languages.

The I18N-Related Tags

To create a multilingual JSTL Web application, you must isolate all language-dependent parts of the program to one area. The language-dependent parts are the strings that your program displays.

To make our forum example multilingual, we must go through the individual source files and find all the strings that are currently in the English language. We then have to move these strings to a resource bundle.

A *resource bundle* is a collection of strings in one particular language, such as English. When you want your program to support languages other than English, you simply translate the resource bundle to another language. These resource bundles are then made available to the Web application. Your program can dynamically choose between resource bundles and change the language that people are using as they work with the application.

A resource bundle is something that exists in standard Java as well as JSTL. In fact, you may have already worked with resource bundles. Later in this chapter, we describe how to create them, but let's begin by explaining the tags that you can use to work with resource bundles. The first tag that we'll look at is the `<fmt:setLocale>` tag, which enables you to select the locale that your Web application is being used from.

Using the `<fmt:setLocale>` Tag

The `<fmt:setLocale>` command is used to set the locale of a Web session. It lets you set the locale for one user or for the entire Web application. Usually, it is preferable to have the Web server maintain individual locales for each of the logged-in users. The locale determines how resource bundles are chosen and how other formatting operations act. There is one form of the `<fmt:setLocale>` tag:

```
<fmt:setLocale value="locale"
[variant="variant"]
[scope="{page|request|session|application}"]/>
```

The `<fmt:setLocale>` tag accepts these attributes:

Attribute	Required	Purpose
value	Y	Specifies a two-part code that represents the ISO-639 language code and an ISO-3166 country code.
scope	N	Specifies the scope of this locale setting. The default is page.
variant	N	Specifies a vendor- or browser-specific variant of the language referenced by value. For more information, refer to the java.util.Locale JavaDocs.

With the `<fmt:setLocale>` tag, you are able to set the current language and country. The scope attribute enables you to specify the locale for either the user or the entire Web application. You accomplish this by specifying one of the standard scopes—page, request, session, or application. Typically, you'll use session scope and set the locale for the current user.

Using the value attribute, you can convey both the language and the country. You separate the two by either a hyphen (-) or an underscore (_). You supply the language code first. This is an ISO-639 code, and you must enter it in lowercase letters (see [Table 10.1](#) for a list of some of the ISO-639 languages). After this, you insert either a hyphen or an underscore and then specify the country code. The country code is an ISO-3166 standard code that you must enter in uppercase letters.

Table 10.1. Some ISO-639 Languages

Language Name	Code	Language Family
ARABIC	ar	SEMITIC
AZERBAIJANI	az	TURKIC/ALTAIC
CZECH	cs	SLAVIC
DANISH	da	GERMANIC
GERMAN	de	GERMANIC
GREEK	el	LATIN/GREEK
ENGLISH	en	GERMANIC
ESPERANTO	eo	INTERNATIONAL AUX.
SPANISH	es	ROMANCE
PERSIAN (Farsi)	fa	IRANIAN
FINNISH	fi	FINNO-UGRIC
IRISH	ga	CELTIC
HEBREW	he	SEMITIC

HINDI	hi	INDIAN
CROATIAN	hr	SLAVIC
HUNGARIAN	hu	FINNO-UGRIC
ICELANDIC	is	GERMANIC
ITALIAN	it	ROMANCE
JAPANESE	ja	ASIAN
JAVANESE	jv	OCEANIC/INDONESIAN
GREENLANDIC	kl	ESKIMO
KOREAN	ko	ASIAN
LATIN	la	LATIN/GREEK
LITHUANIAN	lt	BALTIC
LATVIAN; LETTISH	lv	BALTIC
MACEDONIAN	mk	SLAVIC
MONGOLIAN	mn	[not given]
MOLDAVIAN	mo	ROMANCE
NEPALI	ne	INDIAN
DUTCH	nl	GERMANIC
NORWEGIAN	no	GERMANIC
PUNJABI	pa	INDIAN
POLISH	pl	SLAVIC
PASHTO; PUSHTU	ps	IRANIAN
PORTUGUESE	pt	ROMANCE
ROMANIAN	ro	ROMANCE
RUSSIAN	ru	SLAVIC
SANSKRIT	sa	INDIAN
SINDHI	sd	INDIAN
SLOVAK	sk	SLAVIC
SLOVENIAN	sl	SLAVIC
SERBIAN	sr	SLAVIC
SWEDISH	sv	GERMANIC
SWAHILI	sw	AFRICAN
TAJIK	tg	IRANIAN
THAI	th	ASIAN
TURKISH	tr	TURKIC/ALTAIC
UKRAINIAN	uk	SLAVIC
URDU	ur	INDIAN
UZBEK	uz	TURKIC/ALTAIC
VIETNAMESE	vi	ASIAN

YIDDISH	yi	GERMANIC
YORUBA	yo	AFRICAN
CHINESE	zh	ASIAN
ZULU	zu	AFRICAN

Using the <fmt:setBundle> Tag

The `<fmt:setBundle>` tag loads a resource bundle and stores it in the specified scoped variable. Using this tag, you can load a resource bundle that will be used by subsequent `<fmt:message>` tags. The `<fmt:setBundle>` tag has one form:

```
<fmt:setBundle basename="basename"
[var="varName"]
[scope="{page|request|session|application}"]>
```

These attributes are accepted by the `<fmt:setBundle>` tag:

Attribute	Required	Purpose
<code>basename</code>	Y	Specifies the base name of the resource bundle that is to be loaded.
<code>scope</code>	N	Specifies the scope of the variable contained in the var attribute. The default is page.
<code>var</code>	N	Specifies a scoped variable that will hold the newly instantiated resource bundle.

A resource bundle is a Java class that extends the `java.util.ListResourceBundle` class. The base name specifies the name of the class that implements the base resource bundle, which a program uses when it can't locate any other suitable resource bundle. You use the `basename` attribute to specify the full name of the resource bundle class. In our sample program later in this chapter, the base name of the resource bundle is `com.heaton.bundle.Forum`.

Using this base name allows a program to load the correct resource bundle using the current locale. For example, suppose you have a program that uses the languages English, Spanish, and Chinese. The English translation is stored in the base resource bundle, named `com.heaton.bundle.Forum`. The Spanish translation is stored in the bundle specified by `com.heaton.bundle.Forum_es`, and the Chinese translation is stored in the bundle specified by `com.heaton.bundle.Forum_zh`. The program uses the current locale (referenced in the `<fmt:setLocale>` tag) to determine which resource bundle variant it should load.

The program loads the resource bundle into the scoped variable specified by the `var` attribute. You can specify the scope of this variable by using the `scope` attribute. Once loaded, the resource bundle must be passed to the `<fmt:message>` tag in order for the multilingual text to be displayed.

You have several options when determining the scope. Loading a resource bundle as request or page scope means that the resource bundle will have to be reloaded frequently. Because of this, programmers rarely choose request or page scope.

If you load your resource bundle as application scope, then every user will have access to it. This is a good solution when you've designed your Web site to support several languages, but will have one language specified by the Webmaster. You can define this language in the `web.xml` file so that you can easily specify it as the default language.

Usually, you want to allow your users to pick their own language. To do this, you should use the session scope, which lets you load a different resource bundle for each user. This means that all users can see the Web site content in the language of their choice.

Using the `<fmt:bundle>` Tag

You use the `<fmt:bundle>` tag to specify a resource bundle to be used with the JSTL tags inside the body of the `<fmt:bundle>` and ending `</fmt:bundle>` tags. The `<fmt:bundle>` tag has one form:

```
<fmt:bundle basename="basename"/>
</fmt:setBundle>
```

The `<fmt:setBundle>` tag accepts the following attributes:

Attribute	Required	Purpose
<code>basename</code>	Y	Specifies the base name of the resource bundle that is to be loaded.
<code>prefix</code>	N	Specifies a prefix that will be added to every key attribute specified in the <code><fmt:message></code> tags.

The `<fmt:bundle>` tag will make the specified bundle available to all `<fmt:message>` tags that occur between the bounding `<fmt:bundle>` and `</fmt:bundle>` tags. This saves you the extra step of having to specify the resource bundle for each of your `<fmt:message>` tags. For example, the following two `<fmt:bundle>` blocks would produce the same output:

```
<fmt:bundle basename="com.heaton.bundles.Forum">
    <fmt:message key="login.title"/>
</fmt:bundle>

<fmt:bundle basename="com.heaton.bundles.Forum" prefix="login.">
    <fmt:message key="title"/>
</fmt:bundle>
```

Using the `<fmt:message>` Tag

You use the `<fmt:message>` tag to access the language-specific strings inside a resource bundle. As you construct your multilingual Web applications, you can sprinkle

`<fmt:message>` tags throughout your program in place of language-specific strings. There are three forms of the `<fmt:message>` tag:

```
// Syntax 1: Without body content
<fmt:message key="messageKey"
[bundle="resourceBundle"]
[var="varName"]
[scope="{page|request|session|application}"]/>
// Syntax 2: With a body to specify message parameters
<fmt:message key="messageKey"
[bundle="resourceBundle"]
[var="varName"]
[scope="{page|request|session|application}"]>
<fmt:param> subtags
</fmt:message>
// Syntax 3: With a body to specify key and optional message parameters
<fmt:message [bundle="resourceBundle"]
[var="varName"]
[scope="{page|request|session|application}"]>
key
optional <fmt:param> subtags
</fmt:message>
```

The `<fmt:message>` tag accepts the following attributes:

Attribute	Required	Purpose
<code>bundle</code>	N	Specifies the resource bundle that is to be used. If this tag occurs from within a <code><fmt:bundle></code> tag, then no bundle has to be specified. You could set a default localization context, either by using a <code><setBundle></code> tag or in the configuration file; in either case, you don't have to specify a bundle.
<code>key</code>	N	Specifies the lookup key that identifies the desired string inside the bundle.
<code>scope</code>	N	Specifies the scope of the scoped variable referenced by the <code>var</code> attribute. If the <code>var</code> attribute is not needed, then the <code>scope</code> attribute should not be specified. The default is <code>page</code> .
<code>var</code>	N	Specifies a scoped variable that will receive the value of the string that is being looked up. If no <code>var</code> attribute is specified, then the output will be written to the page.

The first syntax specifies a message that contains no body text. You use the first syntax if you simply want to display a string that contains no parameters. The second syntax allows you to specify parameter values, which we discuss further in the next section. The third syntax lets you use parameter values as well as specify the key of the string, all in the body of the `<fmt:message>` tag. You use the third syntax if you need to dynamically define the key with an embedded tag.

The `<fmt:message>` tag is the most commonly used internationalization tag in JSTL. To create a multilingual Web application, you must insert the `<fmt:message>` tag at every position in which you would normally have language-specific text. This results in a large number of `<fmt:message>` tags. The following code shows how you'd use the `<fmt:message>` tag:

```
<p><fmt:message key="login.note" bundle="${lang}" />
</p>
```

In this code, the `<fmt:message>` tag is freely mixed with other HTML code. Here, we are reading a string from the bundle identified by the scoped variable lang and identified by the key login.note. Both of these values are entirely arbitrary and can be set to any value that you can easily remember. Later in this chapter, we'll show you how to create resource bundles and assign these keys.

It is also possible to copy the string directly from the resource bundle into a scoped variable. You accomplish this by specifying a scoped variable in the var attribute. The value obtained by looking up the key in the resource bundle is copied directly to the variable specified by the var attribute. You can also specify a scope for this variable by using the scope attribute.

Using the `<fmt:param>` Tag

We have just seen how the `<fmt:message>` tag can be used to inject multilingual text in the HTML code of JSP pages. The `<fmt:message>` tag is quite flexible in that you can specify parameters for inclusion in the output. You do this by using the `<fmt:param>` tag, of which there are two forms:

```
// Syntax 1: Value specified via attribute value
<fmt:param value="messageParameter"/>
// Syntax 2: Value specified via body content
<fmt:param>
  body content
</fmt:param>
```

The `<fmt:param>` tag accepts one attribute:

Attribute	Required	Purpose
<code>value</code>	N	Specifies the value that is to be inserted for the parameter.

The first syntax uses the value tag to specify the value that is to be inserted for the parameter. The second syntax uses the body of the tag to specify that value.

To use the `<fmt:param>` tag in your code, you must include parameters in your resource bundle strings. You do this by placing brace-delineated numbers inside your message text. For example, the following string would accept one parameter:

```
Welcome back {0}!
```

This string allows a parameter to be inserted in place of the {0} parameter. To make use of this parameter, use the following code:

```
<fmt:message key="welcome" bundle="${lang}">
  <fmt:param value="${user}" />
</fmt:message>
```

Here, the value stored in the scoped variable user is inserted as the first parameter. Parameter numbers always start with 0. The first `<fmt:param>` tag specified fills in parameter 0; the second fills in parameter 1.

One limitation of JSTL messages is that they are based on the Java class `java.text.MessageFormat`. As a result, you are restricted to only 10 parameters. If you need more than 10, you will have to use multiple tags.

Using the `<fmt:requestEncoding>` Tag

The `<fmt:requestEncoding>` tag is used to specify the encoding type used by forms that post data back to the Web application. There is one form of the `<fmt:requestEncoding>` tag:

```
<fmt:requestEncoding [value="charsetName"] />
```

The `<fmt:requestEncoding>` tag accepts one attribute:

Attribute	Required	Purpose
<code>value</code>	N	Name of character encoding you want to apply when decoding request parameters.

You use the `<fmt:requestEncoding>` tag when you want to specify character encoding for decoding data posted from forms. This tag must be used with character encodings that are different from ISO-8859-1. The tag is required since most browsers do not include a Content-Type header in their requests.

The purpose of the `<fmt:requestEncoding>` tag is to specify the content type of the request. You must specify the content type, even if the encoding of the page generating the response is specified via the `contentType` attribute of a page directive. This is because the response's actual locale (and thus character encoding) may differ from the value specified in the page directive. If the page contains an I18N-capable formatting action that sets the response's locale (and thus character encoding) by calling `ServletResponse.setLocale()`, any encoding specified in the page directive will be overridden.

Resource Bundles

In this section, we show you how to create a resource bundle. You create resource bundles for JSTL the same way you do for ordinary Java applications.

Programmers create resource bundles for a variety of languages. For someone who is already familiar with English, it is quite easy to create a resource bundle for English or one of the other Romance languages. Non-Romance languages, such as the Asian languages, present some additional challenges, however.

What Is a Resource Bundle?

First, we'll describe a resource bundle and show you how to construct one. The method that we focus on involves creating compiled Java classes that extend the `java.util.ListResourceBundle` class. You must compile these class files and make them available to the classpath of your Web application.

A Java resource bundle file contains a series of key-to-string mappings. The following demonstrates a simple Java resource bundle:

```
package com.heaton.bundles;
import java.util.*;

public class Example extends ListResourceBundle {
    public Object[][] getContents() {
        return contents;
    }
    static final Object[][] contents = {
        {"count.one", "One"},
        {"count.two", "Two"},
        {"count.three", "Three"},
    }
}
```

If you compiled this resource bundle and made it available to the classpath, you could use the following JSTL tags to display the three numbers:

```
<fmt:setLocale value="en"/>
<fmt:setBundle basename="com.heaton.bundles.Example" var="lang"
    scope="session"/>
<fmt:message key="count.one" bundle="${lang}"/><br/>
<fmt:message key="count.two" bundle="${lang}"/><br/>
<fmt:message key="count.three" bundle="${lang}"/><br/>
```

The power of resource bundles becomes apparent when you want to translate the site to another language. To accomplish this, you need only provide another resource bundle.

You make no changes to the JSTL code; you only have to specify a different locale. The following code shows how the short resource bundle can be modified for Spanish:

```
package com.heaton.bundles;
import java.util.*;

public class Example_es extends ListResourceBundle {
    public Object[][] getContents() {
        return contents;
    }
    static final Object[][] contents = {
        {"count.one", "Uno"},
        {"count.two", "Dos"},
        {"count.three", "Tres"},
    }
}
```

As you can see, the name of the resource bundle has changed. Previously, the resource name was Example; now the name is Example_es. This is because the code es represents Spanish. Now a user can change the locale to Spanish, and the Spanish strings appear as follows:

```
<fmt:setLocale value="es"/>
```

Creating a Properties File Resource Bundle

It is also possible to create a resource bundle using a properties file. A properties file in Java should contain the same base name as a compiled resource. For example, the forum's English properties file should be stored in a file named forum_en.properties.

The format of a properties file is simple. The entire file consists of lines that are keys to the text data. The following code illustrates a simple properties file:

```
# this is a comment
property1 = value of property 1
property2 = value of property 2
property3 = value of property 3
```

Users access this file in the same way they would a compiled resource. The base name is used to identify the file. The suffix to this file must always be .properties—for example, forum_en.properties.

Creating a Resource Bundle for a Romance Language

In this section, we show you how we created the resource bundles for our forum application. We first look at constructing a resource bundle for a Romance language.

As you may already know, a Romance language is one based on Latin. Languages such as English, French, Italian, and Spanish all fall into this category. Romance languages use

basically the same character set, known as the Latin character set. Because Java uses the Latin character set for source code, it is always easier to deal with a Romance language than a non-Romance language, such as Chinese.

The resource bundles containing the English translation are `Forum.java` and `Forum_en.java`. `Forum.java` is the base translation that a program will use if it does not find the translation in the localized resource bundle. Listing 10.1 shows the English resource bundle used by our forum example.

Listing 10.1 An English Resource Bundle (Forum.java)

```
package com.heaton.bundles;
import java.util.*;

public class Forum extends ListResourceBundle {
    public Object[][] getContents() {
        return contents;
    }
    static final Object[][] contents = {
        {"lang.english", "English"},
        {"lang.spanish", "Spanish"},
        {"lang.chinese", "Chinese"},

        {"button.ok", "OK"},
        {"button.save", "Save"},
        {"button.add", "Add"},
        {"button.cancel", "Cancel"},
        {"button.login", "Login"},
        {"button.register", "Register"},

        {"word.welcome", "Welcome"},
        {"word.password", "Password"},
        {"word.continue", "Continue"},
        {"word.admin", "Administrator"},
        {"word.exit", "Exit"},
        {"word.action", "Action"},
        {"word.user", "User"},
        {"word.type", "Type"},
        {"word.edit", "Edit"},
        {"word.delete", "Delete"},
        {"word.register", "Register"},

        {"type.a","Admin User"},
        {"type.g","Guest User(can't post)" },
        {"type.r","Regular User"},

        {"login.name", "Login Name"},
        {"login.welcome","Welcome to the forum. Please login."},
        {"login.title","Login to Forum."},
        {"login.note","Note: use id of admin and password of admin
on first use. "},
        {"login.badpw","Sorry, we have no one registered with that name."},
```

```

        {"reg.thanks","Thanks for joining the forum. Please choose a
login name and a password. Please enter your password twice to
verify."},
        {"reg.verify","Verify Password"},

        {"reg.needlogin","Must enter a login id!"},
        {"reg.needpw","Must enter a password!"},
        {"reg.badpw","Passwords do not match!"},

        {"welcome.id","Login ID"},

        {"welcome.times","Times on"},

        {"welcome.first","First Login"},

        {"welcome.last","Last Login"},

        {"welcome.failed","Failed Logins"},

        {"welcome.posted","Messages Posted"},

        {"welcome.type","User Type"},

        {"main.welcome","Select one of the following forums. Each
form contains messages on the given topic."},
        {"main.please","Please select a forum."},
        {"main.forumdeleted","Forum Deleted."},

        {"admin.welcome1","From this screen you can edit forums,
delete forums or create new forums."},
        {"admin.welcome2","You may also edit users."},
        {"admin.prompt","The following forums are available:"},
        {"admin.newforum","New Forum"},

        {"admin.editusers","Edit Users"},

        {"editforum.title","Edit Forum"},

        {"editforum.ins1","Forum code is a unique code for this
forum."},
        {"editforum.ins2","Sequence number determines the order
forums will list."},
        {"editforum.code","Forum Code"},

        {"editforum.name","Forum Name"},

        {"editforum.number","Sequence Number"},

        {"newforum.title","Add New Forum"},

        {"editusers.title","Edit Users"},

        {"editusers.ins","The following users exist in your forum."},
        {"editusers.deleted","User deleted."},

        {"edituser.title","User Editor"},

        {"edituser.ins","This page allows you to edit a user. The
only attribute you can change is the user's type."},

        {"forum.none","No messages posted yet."},
        {"forum.post","Post a new message"},

        {"forum.from","From:"},

        {"forum.date","Date:"},

        {"forum.subject","Subject:"},

        {"post.ins","Posting to forum:"},
        {"post.title","Post Message"}};

    
```

}

As you can see in [Listing 10.1](#), every string ever displayed in the forum application has been segregated to this file. The forum example will now refer to this file to get the text when English is the current language.

[Listing 10.2](#) shows the forum application translated to Spanish. I apologize in advance for the translation. It was made through a combination of a Spanish dictionary, a Spanish computer dictionary, and AltaVista's Babelfish. While the translation may not be perfect, it does demonstrate how to load another Romance language as a resource bundle.

TIP

Babelfish, which you can find at <http://world.altavista.com/>, is a useful Internet-based translation service. For a quick prototype translation, you'll find Babelfish useful. However, using AltaVista's translation system on a production system can result in sentences being interpreted incorrectly. Here's an example: "The spirit is willing but the flesh is weak" translated to Spanish gives us, "El alcohol está dispuesto, pero la carne es débil." Translating this Spanish back to English should give the original sentence, yet here's the result: "The alcohol is arranged but the meat is weak." As you can see, the service cannot determine which synonyms to use in place of *spirit* and *flesh*.

Listing 10.2 A Spanish Resource Bundle (Forum_es.java)

```
package com.heaton.bundles;
import java.util.*;

public class Forum_es extends ListResourceBundle {
    public Object[][] getContents() {
        return contents;
    }
    static final Object[][] contents = {
        {"lang.english", "Ingleses"},
        {"lang.spanish", "Español"},
        {"lang.chinese", "Chinois"},

        {"button.ok", "Ok"},
        {"button.save", "Guardar"},
        {"button.add", "Ampliar"},
        {"button.cancel", "Cancelar"},
        {"button.login", "Hacer la conexión"},
        {"button.register", "Registro"},

        {"word.welcome", "Bienvenue"},
        {"word.password", "Contraseña"},
        {"word.continue", "Seguir"},
        {"word.admin", "Operador del Sistema"},
        {"word.exit", "Aalir"},
        {"word.action", "Hacer"},
        {"word.user", "Persona"},
```

```
{"word.type", "Tipo"},  
{"word.edit", "Corrigen"},  
{"word.delete", "Eliminar"},  
{"word.register", "Registro"},  
  
{"type.a","Operador del Sistema"},  
{"type.g","Persona Invitado"},  
{"type.r","Persona Regular"},  
  
{"login.name", "Su nombre de usuario"},  
{"login.welcome","Recepción al foro. Por favor conexión."},  
{"login.title","Recepción al foro. Por favor conexión. "},  
 {"login.note"," Employez l'identification de l'admin et du mot de passe de l'admin sur la première utilisation."},  
 {"login.badpw","Apesadumbrados, tenemos nadie registrados con ese nombre."},  
  
 {"reg.thanks","Gracias por ensamblar el foro. Elija por favor un nombre de la conexión y una palabra de paso. Incorpore por favor su palabra de paso dos veces para verificar."},  
 {"reg.verify","Vérifiez Le Mot de passe"},  
 {"reg.needlogin","Debe incorporar una identificación de la conexión!"},  
 {"reg.needpw","Debe incorporar una palabra de paso!"},  
 {"reg.badpw","Las palabras de paso no corresponden con!"},  
  
 {"welcome.id","Identificación De la Conexión"},  
 {"welcome.times","Épocas encendido"},  
 {"welcome.first","Première Ouverture"},  
 {"welcome.last","Conexión Pasada"},  
 {"welcome.failed","Falladas Conexiones"},  
 {"welcome.posted","Los Mensajes Fijaron"},  
 {"welcome.type","Tipo Del Utilizador"},  
  
 {"main.welcome","Seleccione uno de los foros siguientes.  
Cada forma contiene mensajes en el asunto dado."},  
 {"main.please","Seleccione por favor un foro."},  
 {"main.forumdeleted","Foro Suprimido."},  
  
 {"admin.welcome1","De esta pantalla usted puede corregir foros, suprimir foros o crear foros nuevos."},  
 {"admin.welcome2","Usted puede también corregir a usuarios. "},  
 {"admin.prompt","Los foros siguientes están disponibles: "},  
 {"admin.newforum","Foro Nuevo"},  
 {"admin.editusers","Éditez Les Utilisateurs"},  
  
 {"editforum.title","Corrija El Foro"},  
 {"editforum.ins1","El código del foro es un código único para este foro."},  
 {"editforum.ins2","El número de serie determina los foros de la orden enumerará."},  
 {"editforum.code","Código Del Foro"},  
 {"editforum.name","Nom De Forum"},  
 {"editforum.number","Número De Serie"},
```

```

        {"newforum.title", "Ajoutez Le Nouveau Forum"},

        {"editusers.title", "Corrija A Utilizadores"}, 
        {"editusers.ins", "Los utilizadores siguientes existen en su
foro."}, 
        {"editusers.deleted", "Utilizador suprimido."}, 

        {"edituser.title", "Redactor Del Usuario"}, 
        {"edituser.ins", "Esta paginación permite que usted corrija a
un utilizador. El único atributo que usted puede cambiar es el
tipo del utilizador. "}, 

        {"forum.none", "Ningunos mensajes fijados con todo."}, 
        {"forum.post", "Fije un nuevo mensaje"}, 
        {"forum.from", "De:"}, 
        {"forum.date", "Fecha:"}, 
        {"forum.subject", "Tema:"}, 

        {"post.ins", "Posting to forum:"}, 
        {"post.title", "Mensaje Del Poste"} 

    };
}

```

As you can see in [Listing 10.2](#), every string in the resource bundle has been translated to a Spanish string. This is an ordinary text file that you can edit using Notepad, vi, or another editor.

Creating a Resource Bundle for a Non-Romance Language

We'll now show you how to translate a resource bundle into a non-Romance language. For this example, we use the Chinese language.

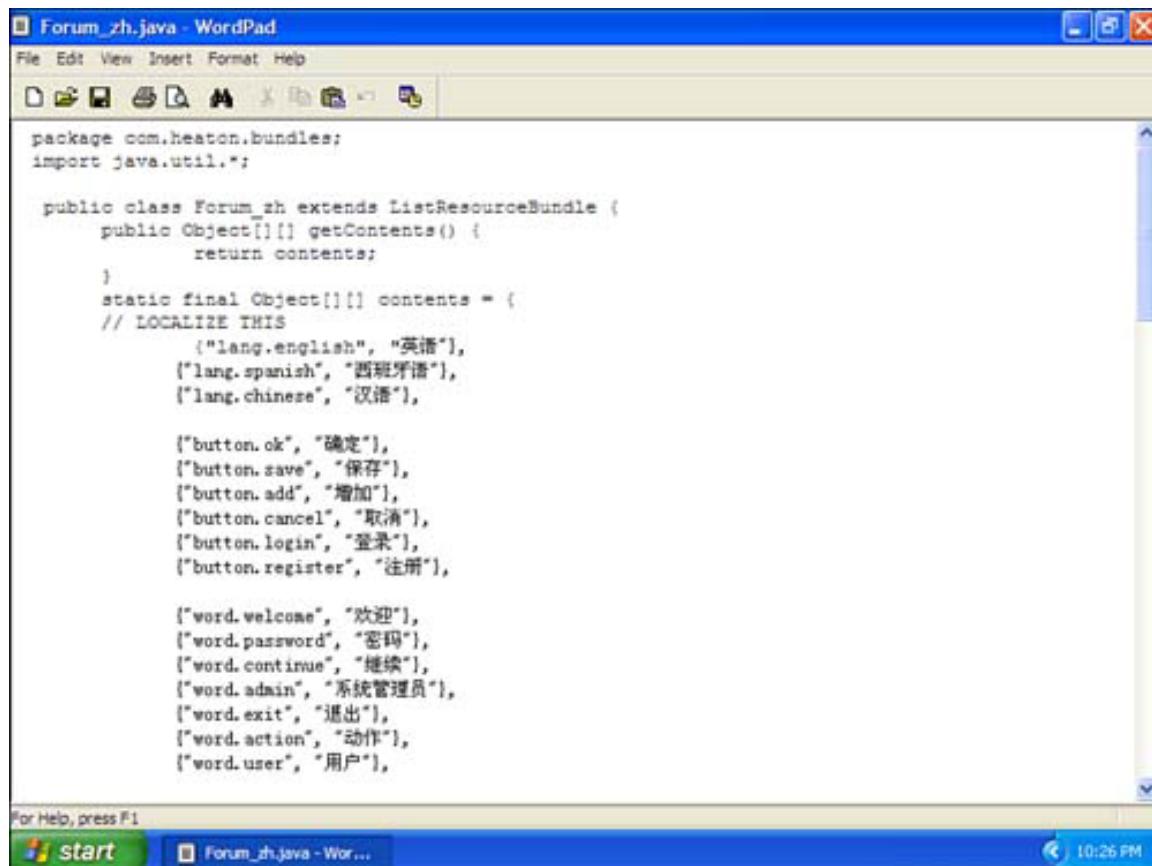
NOTE

The method I used to translate this file to Chinese is typical of how you might translate actual resource bundles. I used a third party; the translation was done by Jenny Yao, a student in one of the Java classes that I teach. I provided Jenny with only the English translation of the Forum.java file. After a couple of emails to verify the correct meanings of some terms, she emailed me a version of my resource bundle translated to Chinese.

This is an important feature of JSTL. It is only necessary to provide your translator with the resource bundle; you do not need to provide any of your Web application's source code. This has several benefits. For example, you don't have to worry about your translator inadvertently changing some of the JSTL code and causing an error. In addition, keeping your JSTL code inside your company and not disclosing it to third-party translation services adds a layer of security.

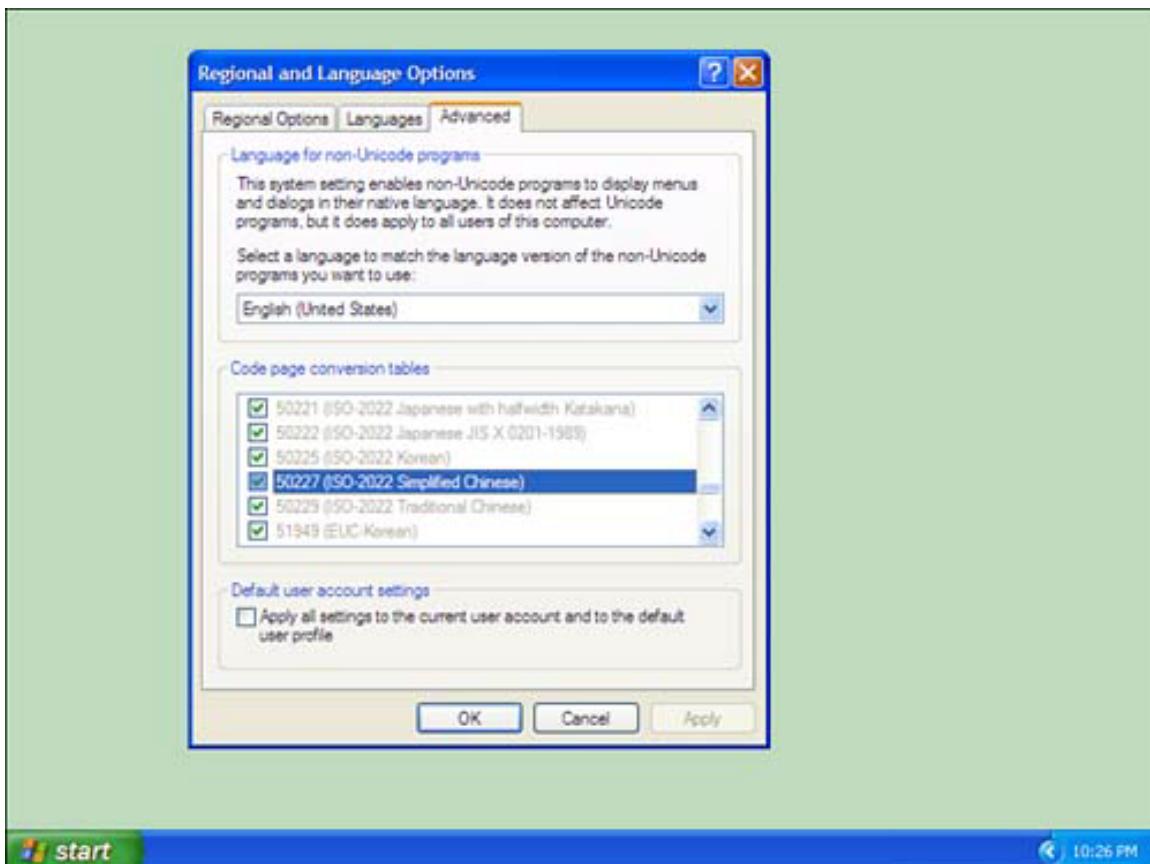
The Chinese version of the resource bundle is contained in the file Forum_zh.java. You cannot simply open this file with Notepad to view it. If you're using Windows, we suggest that you use WordPad to view the file, as shown in Figure 10.1. You can obtain this file from the Sams Web site.

Figure 10.1. Viewing the Chinese resource bundle.



It is also important to note that you must have the Chinese fonts installed in order for WordPad to display your file correctly. In Windows XP, select Settings, Control Panel, Regional and Language Options. In the resulting dialog box, click the Advanced tab and select the option Simplified Chinese, as shown in [Figure 10.2](#).

Figure 10.2. Installing the Chinese character set.



Compiling the Resource Bundles

Now that we've created four resource bundles to use with our forum application, we must compile the bundles. Listing 10.3 contains a simple build script named build.sh (build.bat for Windows) that will build these files. The same script file will work under either Windows or Unix.

Listing 10.3 A Resource Bundle Build Script (build.sh)

```
javac -encoding UTF-16 ./com/heaton/bundles/Forum_zh.java
javac ./com/heaton/bundles/Forum_es.java
javac ./com/heaton/bundles/Forum.java
javac ./com/heaton/bundles/Forum_en.java
jar cvf bundle.jar ./com/heaton/bundles/*.class
```

As you can see, the Chinese file, Forum_zh.java, is given special handling. This is because it cannot be compiled by Java unless we specify the encoding. For this file, we used UTF-16 encoding, as revealed in the first line of Listing 10.3.

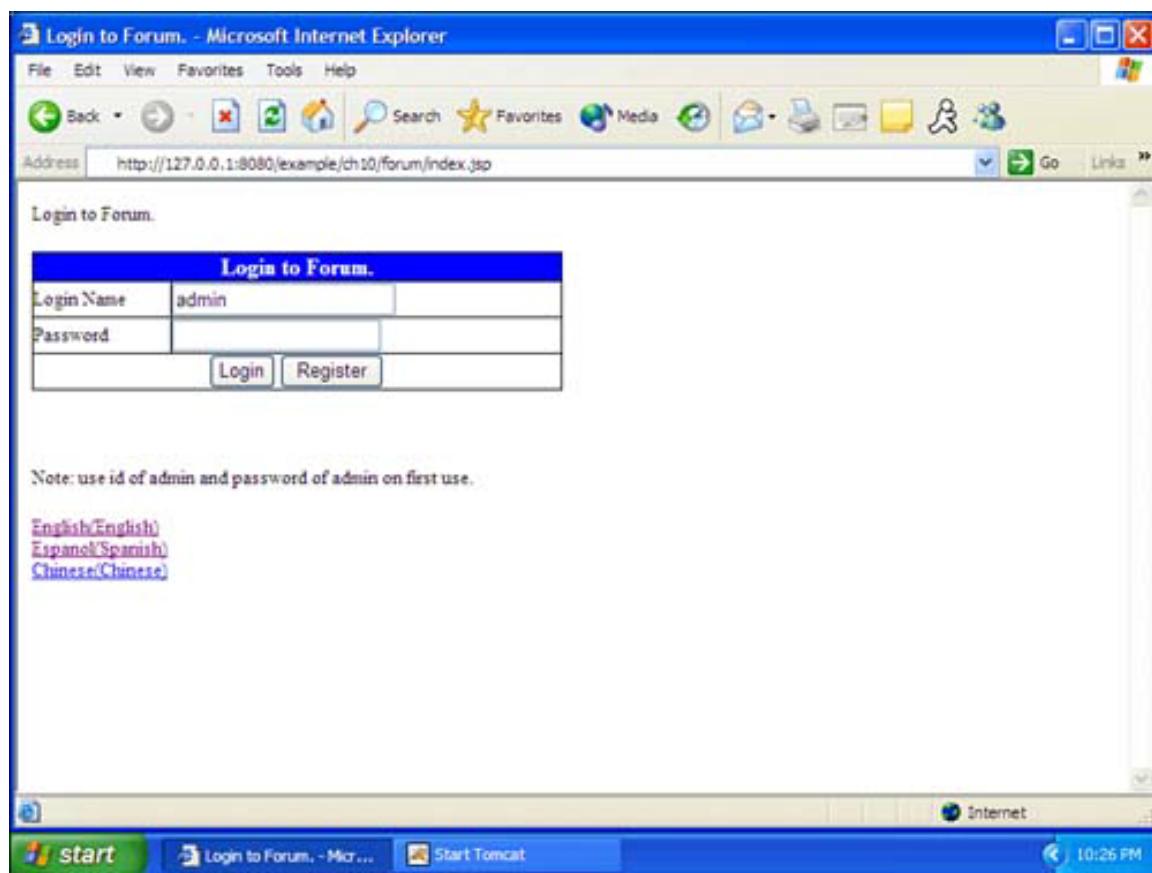
If all goes well, this script creates a file named bundle.jar, which you should place in Tomcat's lib directory so that it is exposed to the classpath. You can download a zipped version of the Tomcat root directory from the Sams Web site; this version has all of the files in the correct locations for every example in this book.

A Multilingual Forum Application

You should now have the necessary resource bundles for our forum application. Let's see how to convert the forum example shown in [Chapter 7](#) to a multilingual application. The majority of the work involves changing all the English text to `<fmt:message>` tags. We won't show you every converted JSP file; rather, we'll just examine the `index.jsp` file in order to demonstrate how to use the `<fmt:message>` tags.

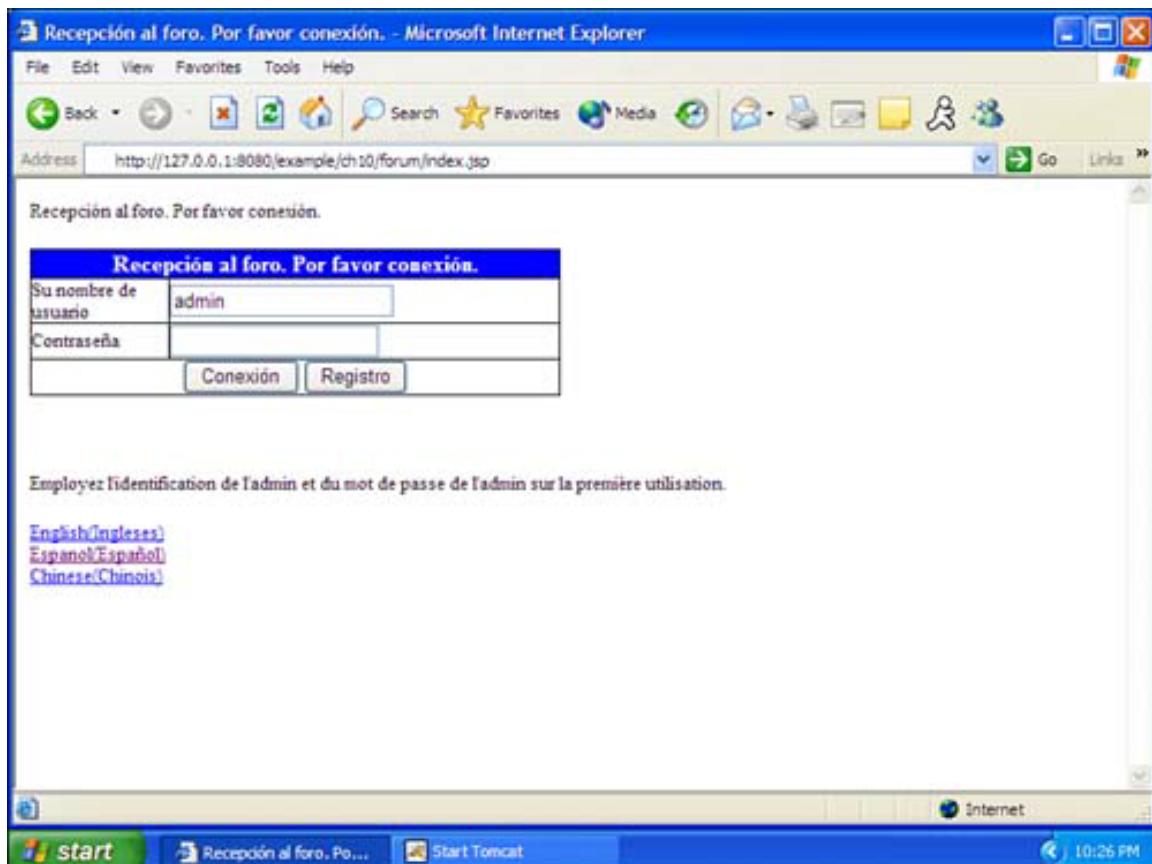
The index page for our forum application now allows you to easily switch between the three languages. The default language, English, is shown in [Figure 10.3](#).

Figure 10.3. Our multilingual form in English.



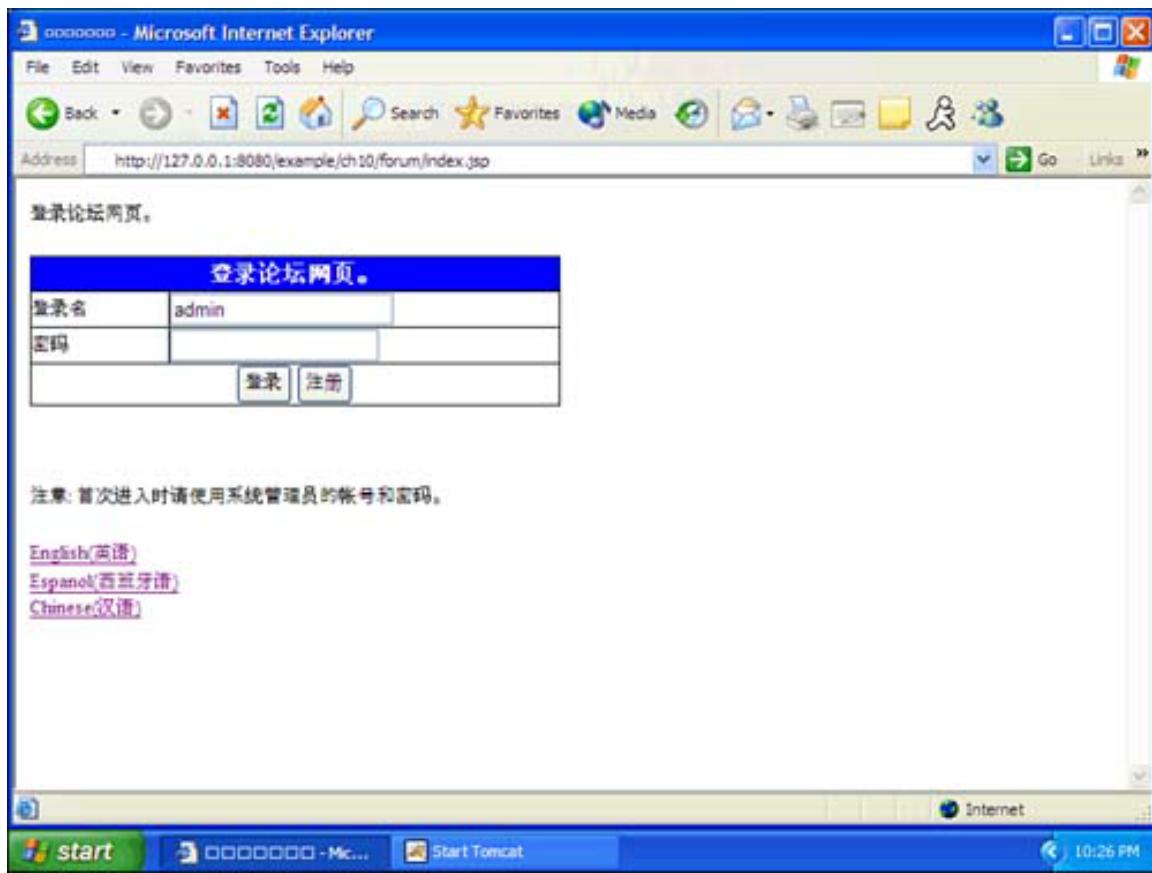
Simply clicking on the Spanish link will cause the site to redisplay in Spanish. [Figure 10.4](#) shows this transformation.

Figure 10.4. The forum in Spanish.



If you click the Chinese link, the site will be displayed in Chinese. You'll notice that the Chinese characters display just fine in the text regions and buttons. However, those characters do not display properly in the browser's title bar. This is because very few browsers can correctly display Chinese characters contained in the `<title></title>` tags of a page. Romance languages will display just fine. Because of this, you may want to replace all the HTML `<title>` tags with generic English titles that will not be translated. [Figure 10.5](#) shows the example running in Chinese.

Figure 10.5. The forum in Chinese.



Now that you've seen how the forum looks in each of the three languages, let's examine how this was actually done. The only file that changes significantly is the index.jsp file. The other files change only in that they now specify UTF-8 as the character encoding, and they use `<fmt:format>` to display strings. Listing 10.4 shows the index.jsp file from Chapter 7, rewritten for multilingual processing.

Listing 10.4 A Multilingual index.jsp (index.jsp)

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt" %>
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jstl/core-rt" prefix="c-rt" %>
<%@ taglib uri="http://java.sun.com/jstl/sql" prefix="sql" %>

<c:if test="${dataSource==null}">
    <sql:setDataSource var="dataSource" driver="org.gjt.mm.mysql.
    Driver"
    url="jdbc:mysql://localhost/forum?user=forumuser"
    scope="session" />
</c:if>

<c:if test="${lang==null}">
    <fmt:setBundle basename="com.heaton.bundles.Forum" var="lang"
    scope="session"/>
```

```

</c:if>

<c:if test="${param.lang!=null}">
    <fmt:setLocale value="${param.lang}" />
    <fmt:setBundle basename="com.heaton.bundles.Forum" var="lang"
scope="session"/>
    <c:redirect url="index.jsp"/>
</c:if>

<c:if test="${pageContext.request.method=='POST' }">
    <c:if test="${param.reg!=null}">
        <c:redirect url="register.jsp" />
    </c:if>

    <sql:query var="users" dataSource="${dataSource}">select c_
uid,c_type
from t_users where c_uid = ? and c_pwd = ?
<sql:param value="${param.uid}" />

<sql:param value="${param.pwd}" />
</sql:query>

<c:choose>
    <c:when test="${users.rowCount<1}">
        <h3 color="red"><fmt:message key="login.badpw"
bundle="${lang}"/></h3>

        <sql:update var="result" dataSource="${dataSource}">update
t_users set c_bad = c_bad + 1 where c_uid = ?
<sql:param value="${param.uid}" />
</sql:update>
    </c:when>

    <c:otherwise>
        <%
            Cookie mycookie = new Cookie("login",request.
getParameter("uid"));
            mycookie.setMaxAge(0x7fffffff);
            response.addCookie(mycookie);
        %>
        <c:forEach var="aUser" items="${users.rows}">
            <c:set var="userID" value="${aUser.c_uid}"
scope="session" />

            <c:set var="userType" value="${aUser.c_type}"
scope="session" />
        </c:forEach>

        <c:redirect url="welcome.jsp" />
    </c:otherwise>
</c:choose>
</c:if>

<c-rt:if test="<%=request.getCookies() !=null%">
    <c-rt:forEach var="aCookie" items="<%=request.getCookies()%">
        <c:if test="${aCookie.name=='login'}">
            <c:set var="uid" value="${aCookie.value}" />

```

```

        </c:if>
    </c-rt:forEach>
</c-rt:if>

<html>
    <head>
        <title><fmt:message key="login.title" bundle="${lang}"/></title>
    </head>

    <body>
        <fmt:message key="login.welcome" bundle="${lang}"/><br>
        <form method="POST">
            <table border="1" cellpadding="0" cellspacing="0"
                style="border-collapse: collapse" bordercolor="#111111"
                width="49%" id="AutoNumber1">
                <tr>
                    <td width="100%" colspan="2" bgcolor="#0000FF">
                        <p align="center">
                            <b>
                                <font color="#FFFFFF" size="4">
                                    <fmt:message key="login.title" bundle="${lang}"/>
                                </font>
                            </b>
                        </p>
                    </td>
                </tr>

                <tr>
                    <td width="26%"><fmt:message key="login.name"
bundle="${lang}"/></td>

                    <td width="74%">
                        <input type="text" name="uid" value="" size="20" />
                    </td>
                </tr>

                <tr>
                    <td width="26%">
                        <fmt:message key="word.password" bundle="${lang}"/></td>

                    <td width="74%">
                        <input type="password" name="pwd" size="20" />
                    </td>
                </tr>

                <tr>
                    <td width="100%" colspan="2">
                        <p align="center">
                            <input type="submit" value="" name="Login" />
                            <input type="submit" value="" name="reg" />
                        </p>
                    </td>
                </tr>
            </table>
        </form>
    </body>
</html>

```

```

        </tr>
    </table>

    <p>&#160;</p>
</form>

<p><fmt:message key="login.note" bundle="${lang}" />
</p>

    <a href="index.jsp?lang=en">English(<fmt:message
key="lang.english" bundle="${lang}" />)</a><br>
    <a href="index.jsp?lang=es">Espanol(<fmt:message
key="lang.spanish" bundle="${lang}" />)</a><br>
    <a href="index.jsp?lang=zh">Chinese(<fmt:message
key="lang.chinese" bundle="${lang}" />)</a><br>
</body>
</html>

```

Notice the first line of the file:

```
<%@ page contentType="text/html; charset=UTF-8" %>
```

This line specifies that the file is using UTF-8 encoding. Without this line, the Chinese characters will not display correctly. To switch between the languages, we've provided several hyperlinks near the bottom that will take you back to the index file with the language value contained in the parameter param. This will allow the following code to correctly change the language:

```

<c:if test="${lang==null}">
    <fmt:setBundle basename="com.heaton.bundles.Forum"
    var="lang" scope="session"/>
</c:if>

```

First, we check to see whether any language bundle is loaded. This handles the default first view when English is selected. The above line loads the bundle into the scoped variable lang:

```

<c:if test="${param.lang!=null}">
    <fmt:setLocale value="${param.lang}" />
    <fmt:setBundle basename="com.heaton.bundles.Forum" var="lang"
scope="session"/>
    <c:redirect url="index.jsp"/>
</c:if>

```

Next, we check to see whether a new language was specified. If a language was specified, we set it to the current locale. We then reload the resource bundle, reflecting the change in locale. Finally, to make the browser redisplay the index page, we redirect to that page using a `<c:redirect>` tag.

Summary

In this chapter, we saw how you can add languages to a Web application. We extended the sample program from [Chapter 7](#) to support Spanish and Chinese. Our forum application is now a multilingual Web application.

In the next chapter, we make one final change to our forum application. We take the many SQL commands that exist in the application and isolate them to a custom tag library. You'll learn how you can use your own custom tag libraries, in conjunction with JSTL, to create Web applications.

Chapter 11. Creating Your Own Tag Libraries

In this chapter, we show you how to create custom tag libraries. We focus particularly on creating tag libraries that work well with the JSTL tag library. By designing your tag libraries so that they can deal with JSTL scoped variables and the JSP 1.3 expression language (EL), you will be able to create tag libraries that easily integrate with JSTL code.

Tag libraries allow you to take procedures that would be lengthy to write using just JSTL and encapsulate them in a tag. One perfect example of this is the JSTL SQL tags. It is generally not a good idea to put SQL database code in JSP pages. Database code is an implementation detail that should be separated from the presentation-oriented JSP pages. The example we present in this chapter shows you how to take our forum example from [Chapter 10](#), "Understanding JSTL Internationalization," and isolate all the database logic behind a custom tag library. We begin with a discussion of how we should structure the tag library.

Developing Custom Tag Libraries

While our goal in this chapter is to develop a custom tag library that encapsulates the database management of our forum application, this is not to say that we will completely replace the use of JSTL. Rather, we want to design a number of custom tags that are capable of processing forum information. We will still use JSTL for formatting, iteration, flow, and other basic presentation-level issues.

It is important to correctly spread the application across the necessary tags. This requires some design to determine the best mix of classes and tags that will be used by your JSP pages. In the next section, we examine the tags that we created for our forum application.

Tags Used by the Forum Application

The forum application's tag library consists of 16 tags. These tags are summarized in [Table 11.1](#).

Table 11.1. Custom Tags Used by the Forum Application

Tag Name	Attributes	Purpose
<code>createForum</code>	<code>code, name, sequence</code>	Creates a new forum.
<code>deleteForum</code>	<code>code</code>	Deletes an existing forum.
<code>deleteMessage</code>	<code>code, number</code>	Deletes a forum message.
<code>deleteUser</code>	<code>id</code>	Deletes a user.
<code>editForum</code>	<code>code, name, sequence</code>	Edits a forum.
<code>editUser</code>	<code>id, type</code>	Edits a user.

<code>expire</code>		Checks to see if the login has expired.
<code>listForums</code>	<code>scope, var</code>	Lists all of the forums.
<code>listMessages</code>	<code>code, scope, var</code>	Lists all of the messages in a forum.
<code>listUsers</code>	<code>var</code>	Lists all of the users.
<code>loadForum</code>	<code>code, scope, var</code>	Loads a forum.
<code>loadUser</code>	<code>id, scope, var</code>	Loads a user.
<code>loginUser</code>	<code>id, password, scope, var</code>	Logs in the specified user.
<code>newUser</code>	<code>id, password, scope, var</code>	Creates a new user.
<code>postMessage</code>	<code>code, from, message, subject</code>	Posts a new message.
<code>saveUser</code>	<code>var</code>	Saves the specified user.

These tags are used throughout the JSP pages that make up our forum application. The structure of the JSP pages remains the same. The only difference is that each of these JSP pages uses the tags listed in [Table 11.1](#) rather than the JSTL SQL tags to process information.

If you examine the tags in [Table 11.1](#), you'll notice that they map closely to the SQL queries and updates performed by our forum application. This allows implementation details to be separated from the presentations. For example, when a user first logs in, the last login, number of failed logins, and number of connections are all updated. In our forum application, this update task was performed by a SQL UPDATE command that we embedded into the login page. Now we have a tag called `<forum:loginUser>` that performs this operation. The JSP programmer simply calls `<forum:loginUser>` and the user is logged in. The JSP programmer does not need to be concerned with what data is updated behind the scenes as the `<forum:loginUser>` tag is executed.

When the `<forum:loginUser>` tag is called and a successful login occurs, an object is returned that contains the information specific to the user who just logged in. In this case, a class named User is returned that contains the user. The other tags used by the forum application return data in a variety of ways. Some return simple strings or numbers, but most use three classes that we created to hold the data for users, forums, and messages. We examine these data classes in the next section.

For this application, we chose to use a number of tags that process data in the form of several data classes. In the next sections, we discuss the overall structure of the forum application.

Data Structures Used by the Forum Application

As you will recall from [Chapter 7](#), "Accessing Data with SQL Tags," three primary tables make up our forum application. These three tables store the users, forums, and messages.

To implement the tag library, we need to create three Java classes capable of holding each of these three record types. The first of these is the User class.

Storing Users

The forum application stores basic information about the users that access it. This data is stored in the t_users table. When the forum application uses the tag library to access data from the t_users table, it returns records using the User class, shown in [Listing 11.1](#).

Listing 11.1 The User Class (User.java)

```
package com.heaton.forum;
import java.util.*;  
  
public class User {  
  
    /**  
     * The login id of this user.  
     */  
    private String id;  
  
    /**  
     * The password for this user.  
     */  
    private String password;  
  
    /**  
     * The number of times the user has accessed the forum.  
     */  
    private int accesses;  
  
    /**  
     * The first time time the user has connected.  
     */  
    private Date first;  
  
    /**  
     * The last time the user has logged on.  
     */  
    private Date last;  
  
    /**  
     * The number of failed logins for this user.  
     */  
    private int bad;  
  
    /**  
     * The number of messages posted.  
     */  
    private int posted;  
  
    /**  
     * The type of this user.  
     * A = admin  
    }
```

```
* R = regular
* G = guest
*/
private String type;

public String getId()
{
    return this.id;
}

public void setId(String id)
{
    this.id = id.toLowerCase();
}

public String getPassword()
{
    return this.password;
}

public void setPassword(String password)
{
    this.password = password;
}

public int getAccesses()
{
    return this.accesses;
}

public void setAccesses(int accesses)
{
    this.accesses = accesses;
}

public Date getFirst()
{
    return this.first;
}

public void setFirst(Date first)
{
    this.first = first;
}

public Date getLast()
{
    return this.last;
}

public void setLast(Date last)
{
    this.last = last;
}

public int getBad()
```

```

{
    return this.bad;
}

public void setBad(int bad)
{
    this.bad = bad;
}

public int getPosted()
{
    return this.posted;
}

public void setPosted(int posted)
{
    this.posted = posted;
}

public String getType()
{
    return this.type;
}

public void setType(String type)
{
    this.type = type;
}
}

```

[Listing 11.1](#) contains an ordinary Java data class that has one field property for each of the fields in the underlying table.

Storing Forums

The t_forums table stores the individual forums where the users post and read messages. When the forum application uses the tag library to access data from the t_forums table, it returns records using the Forum class, shown in [Listing 11.2](#).

Listing 11.2 The Forum Class (Forum.java)

```

package com.heaton.forum;
import java.util.*;

public class Forum {

    /**
     * The code that identifies this forum.
     */
    private String code;

    /**
     * The name of this forum

```

```

        */
private String name;

/**
 * The sequence number of this forum.
 * Forum sequence numbers are used to
 * determine the order that the forums are
 * listed in.
 */
private int sequence;

public String getCode()
{
    return this.code;
}

public void setCode(String code)
{
    this.code = code.toUpperCase();
}

public String getName()
{
    return this.name;
}

public void setName(String name)
{
    this.name = name;
}

public int getSequence()
{
    return this.sequence;
}

public void setSequence(int sequence)
{
    this.sequence = sequence;
}
}

```

This structure represents only the forums, or message areas. The Forum class does not store individual messages. All individual messages are stored in a common table. Let's now examine the Java class used to hold individual messages.

Storing Messages

A forum may have many different messages. These messages are stored in the t_messages table. The individual records from this table are stored in instances of the Message class, shown in Listing 11.3.

[Listing 11.3 Storing a Message \(Message.java\)](#)

```
package com.heaton.forum;
import java.util.*;

public class Message {

    /**
     * The forum code that designates the forum
     * that this message belongs to.
     */
    private String forumCode;

    /**
     * The message number.
     */
    private int number;

    /**
     * When this message was posted.
     */
    private Date posted;

    /**
     * The subject of this message.
     */
    private String subject;

    /**
     * The sender of this message.
     */
    private String sender;

    /**
     * The message body.
     */
    private String message;

    public String getForumCode()
    {
        return this.forumCode;
    }

    public void setForumCode(String forumCode)
    {
        this.forumCode = forumCode.toUpperCase();
    }

    public int getNumber()
    {
        return this.number;
    }

    public void setNumber(int number)
    {
        this.number = number;
    }

    public Date getPosted()
```

```

{
    return this.posted;
}

public void setPosted(Date posted)
{
    this.posted = posted;
}

public String getSubject()
{
    return this.subject;
}

public void setSubject(String subject)
{
    this.subject = subject;
}

public String getSender()
{
    return this.sender;
}

public void setSender(String sender)
{
    this.sender = sender;
}

public String getMessage()
{
    return this.message;
}

public void setMessage(String message)
{
    this.message = message;
}
}

```

The Components of a Tag Library

You have now seen the basic structure of the forum application's tag library. We must now implement this tag library and modify the JSP files to use the new tag library rather than the SQL tags we used in [Chapter 10](#).

In essence, a tag library is a JAR and an XML-formatted tag library descriptor (TLD) configuration file. Installing a tag library consists of three steps:

1. Copy the tag library descriptor file (*.tld) to the `WEB-INF` directory for your Web application.

2. Modify the `web.xml` file for your Web application so that it includes the TLD file that you copied in step 1.
3. Copy the tag library's JAR file to the `/lib` directory of your Web application so that your Web server can access it.

Let's now examine each of these steps in detail. In particular, we focus on the structure of the JAR and TLD files so that you will be able to create your own tag libraries that you can use to supplement JSTL with your own company-specific procedures.

The Tag Library Descriptor File (TLD)

The forum application's tag library stores its TLD information in a file called `forum.tld`. This file is quite lengthy, and will not be included here in its entirety. It contains definitions for each of the 16 tags that make up the forum tag library.

The TLD File Header

The `forum.tld` file begins with basic header information:

```
<?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/dtd/web-jsptaglibrary_1_2.dtd">
<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name>forum</short-name>
  <uri>http://java.jeffheaton.com/taglib/jstl/forum</uri>
  <display-name>Forum Example</display-name>
  <description>An example tag library used to implement a
forum.</description>
```

This header contains basic information about the forum tag library and what sort of environment it expects. This information should be made available for any TLD file that you create. One of the most important elements is the `<uri>` tag. This identifier enables the JSP pages to use this tag library. As you may recall from [Chapter 1](#), "Understanding JSP Custom Tags," we used the following taglib directive:

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

As you can see, a URI is specified. This URI corresponds to a URI specified in the TLD for the core JSTL tag library. If you wanted to use the forum tag library, you must use a similar taglib directive, except you insert the forum URI:

```
<%@ taglib uri="http://java.jeffheaton.com/taglib/jstl/forum"
prefix="forum" %>
```

The URI is not an actual Web address. If you copy and paste a URI from a tag library into a browser, you may or may not see anything of meaning. The URI is simply a

common identifier that is used to tie the JSP pages to the appropriate TLD file. Next, we look at the tag definitions that follow the header.

Tag Definitions

After the header information, you see `<tag>` tags that specify the individual tags in the library. This tag is defined in our forum application tag library:

```
<tag>
  <name>loadUser</name>
  <tag-class>com.heaton.forum.LoadUserTag</tag-class>
  <body-content>empty</body-content>
  <description>Used to load a user.</description>
  <attribute>
    <name>var</name>
    <required>true</required>
    <rtpexprvalue>false</rtpexprvalue>
  </attribute>
  <attribute>
    <name>id</name>
    <required>true</required>
    <rtpexprvalue>false</rtpexprvalue>
  </attribute>
  <attribute>
    <name>scope</name>
    <required>false</required>
    <rtpexprvalue>false</rtpexprvalue>
  </attribute>
</tag>
```

This tag defines the `<forum:loadUser>` tag. Each tag consists of two parts. First, each tag has header information of its own. This includes the tag's name and other important information. Following the tag's header information are several attribute tags. If no attributes are used with the tag, then this section is empty.

[Table 11.2](#) describes the tag header elements.

Table 11.2. Tag Header Elements

Element Name	Purpose
<code>attribute</code>	There will be zero or more such elements. They specify the attributes that can be used with this tag. Table 11.3 defines the elements that make up an attribute node.
<code>body-content</code>	Specifies the type of body that this tag has. If the tag has no body, the value <code>empty</code> should be used. If other JSP code can be contained in the body, the value <code>jsp</code> should be used.
<code>description</code>	A text description that defines what this tag does.
<code>name</code>	The name of the tag. This is the name the JSP code will use when referring to this tag.

<code>tag-class</code>	The class that implements the tag.
------------------------	------------------------------------

The attribute element contains several subelements. This allows you to include information about each of the attributes. [Table 11.3](#) summarizes these subelements.

Table 11.3. Tag Attribute Subelements	
Element Name	Purpose
<code>name</code>	The name of the attribute.
<code>required</code>	A true/false value that specifies whether this tag is required. If the tag is specified as required and the tag is not provided, an error will be raised before your tag is executed.
<code>rtpexprvalue</code>	Allows RT expressions to be used for the value of the attribute. RT expressions are expressions such as <code><%=request.getMethod()%></code> .

You'll notice that the tags do not accept RT expressions. Consider the following tag:

```
<testlib:printIt value="<%="10*10%" />
```

If you had specified a value of false for the rtpexprvalue attribute, your tag would get the value `<%=10*10%>` when the value attribute was retrieved. If a value of true had been specified for the rtpexprvalue attribute, the value 100 would be retrieved. The tags we are using in this example are designed to use EL expressions and not RT expressions.

Whenever possible, you should use EL expressions rather than RT expressions. EL expressions are an important part of JSP 1.3 and offer more flexibility than the older RT expressions. Because of this you will usually set the rtpexprvalue attribute to false. The forum tag library tags do not make use of RT expressions. The tag that we just examined could be expressed using an EL expression as follows:

```
<testlib:printIt value="${10*10}" />
```

Setting the rtpexprvalue attribute to true will not help you process EL expressions. Processing EL expressions is important if you want your tag library to be used in conjunction with the JSTL tag libraries. We discuss processing EL expressions later in this chapter.

Updating web.xml for Custom Tags

Now that you have created a TLD file for your tag library, you must modify your web.xml file so that it uses the TLD file. [Listing 11.4](#) shows the web.xml file for our forum application.

Listing 11.4 The web.xml File

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

<context-param>
    <param-name>javax.servlet.jsp.jstl.fmt.timeZone</param-name>
    <param-value>US/Central</param-value>
</context-param>

<context-param>
    <param-name>database-driver</param-name>
    <param-value>org.gjt.mm.mysql.Driver</param-value>
</context-param>

<context-param>
    <param-name>database-url</param-name>
    <param-value>
        jdbc:mysql://localhost/forum?user=forumuser</param-value>
    </param-value>
</context-param>

<taglib>
    <taglib-uri>http://java.sun.com/jstl/fmt</taglib-uri>
    <taglib-location>/WEB-INF/fmt.tld</taglib-location>
</taglib>

<taglib>
    <taglib-uri>http://java.sun.com/jstl/fmt-rt</taglib-uri>
    <taglib-location>/WEB-INF/fmt-rt.tld</taglib-location>
</taglib>

<taglib>
    <taglib-uri>http://java.sun.com/jstl/core</taglib-uri>
    <taglib-location>/WEB-INF/c.tld</taglib-location>
</taglib>

<taglib>
    <taglib-uri>http://java.sun.com/jstl/core-rt</taglib-uri>
    <taglib-location>/WEB-INF/c-rt.tld</taglib-location>
</taglib>

<taglib>
    <taglib-uri>http://java.sun.com/jstl/sql</taglib-uri>
    <taglib-location>/WEB-INF/sql.tld</taglib-location>
</taglib>

<taglib>
    <taglib-uri>http://java.sun.com/jstl/sql-rt</taglib-uri>
    <taglib-location>/WEB-INF/sql-rt.tld</taglib-location>
</taglib>

<taglib>
    <taglib-uri>http://java.sun.com/jstl/x</taglib-uri>
    <taglib-location>/WEB-INF/x.tld</taglib-location>
</taglib>
```

```

<taglib>
  <taglib-uri>http://java.sun.com/jstl/x-rt</taglib-uri>
  <taglib-location>/WEB-INF/x-rt.tld</taglib-location>
</taglib>

<taglib>
  <taglib-uri>
    http://java.jeffheaton.com/taglib/jstl/forum</taglib-uri>
    <taglib-location>/WEB-INF/forum.tld</taglib-location>
  </taglib>

</web-app>

```

The file in [Listing 11.4](#) contains several important pieces of information. The `<taglib>` elements are used to specify configuration information to be used with the forum tag library. Let's look at the `<taglib>` statements next.

Including Tag Libraries in web.xml

Each of the `<taglib>` elements that you see in [Listing 11.4](#) is a tag library that we are making available. Most of the `<taglib>` elements are used to include JSTL tag libraries. This element is used to include the forum application's tag library:

```

<taglib>
  <taglib-uri>
    http://java.jeffheaton.com/taglib/jstl/forum</taglib-uri>
    <taglib-location>/WEB-INF/forum.tld</taglib-location>
  </taglib>

```

As you can see, we've provided a path to the TLD file. The URI is also specified again with the `<taglib-uri>` element. By including this taglib node in your `web.xml` file, you make the tag library available.

Of course, taglib directives are not the only thing stored in the `web.xml` file. You can also store configuration information about your application.

Storing Configuration Information in web.xml

The forum example in [Chapter 7](#) contained two `index.jsp` files. We configured the main `index.jsp` file to work with the MySQL database, and we configured the secondary `index-msa.jsp` file to work with Microsoft Access. This is no longer necessary now that we are using a custom tag library. In the version of the forum example presented in this chapter, we store information about the database connection in the `web.xml` file, and allow our tag library and JSP files to remain free of ties to Microsoft Access or MySQL.

Configuration information stored in the `web.xml` file is contained in `<context-param>` tags. The following two `<context-param>` tags are used to store information about the database we'll use:

```

<context-param>
    <param-name>database-driver</param-name>
    <param-value>org.gjt.mm.mysql.Driver</param-value>
</context-param>

<context-param>
    <param-name>database-url</param-name>
    <param-value>
        jdbc:mysql://localhost/forum?user=forumuser</param-value>
    </param-value>
</context-param>

```

A context tag named database-url provides the JDBC URL of the database we plan to access. A second context tag, named database-driver, is used to specify the database driver that we'll use to connect to the database. This code is configured to work with a MySQL database. If you want the example to work with the Microsoft Access database provided in [Chapter 7](#), you have to modify the code as follows:

```

<context-param>
    <param-name>database-driver</param-name>
    <param-value>sun.jdbc.odbc.JdbcOdbcDriver</param-value>
</context-param>

<context-param>
    <param-name>database-url</param-name>
    <param-value>
        jdbc:odbc:forum</param-value>
    </param-value>
</context-param>

```

This code assumes that you are using the DSN forum as specified in [Chapter 7](#). The values stored in `<context-param>`, which is located in `web.xml`, can easily be accessed from a tag library. We see how this is done in the next section.

Creating the Tag Classes

So far, we've explained how to set up the configuration files that go with the tag libraries, but we haven't yet seen how to create the tag libraries. If you examine the files that make up the forum tag library, you'll notice that there are essentially three classes of files:

- Tag files of the form `xxxTag.java`, such as `LoadUserTag.java`
- Data classes such as `Forum.java` or `User.java`
- The session management file, `Session.java`

We have already described the data classes. Next, we examine session management and the tag files.

Handling Sessions

Much of the functionality for our forum application's tag library is contained in the `Session` class. The `Session` class is used to ensure that each user session uses only one

database connection. It would be inefficient for each user session to create and dispose of database connections for each SQL command issued.

One potential hazard of this approach is that database connections will remain open even after the user has closed the browser. Once the browser is closed, it will be 30 minutes (or whatever session timeout is set to) until the session is closed. In a very high-volume Web site, this could lead to high resource usage.

The best way to handle such an issue is to use a connection pool. Implementing a JDBC connection pool is beyond the scope of this book. You may want to look at a prebuilt JDBC connection pool, such as struts GenericDriver.

Note that the Session class is not a required component of a tag library. It is simply a utility class that we created to encapsulate some of the common functionality required by all of the forum application's tags. Because every tag in the forum example makes use of the Session class, let's begin by examining the Session class.

The Session class works by creating a new Session object for each user session. This is done by storing the Session object as a session-scoped attribute.

Obtaining a New Session

To obtain a Session object, the static method `getSession()` should be called from the Session object:

```
/**  
 * Factory method to create sessions. This method  
 * will ensure that there is only one object created  
 * per user session.  
 *  
 * @param ctx The page context.  
 * @return A Session object.  
 * @exception JspException Thrown on general error.  
 */  
static Session getSession(PageContext ctx)  
throws JspException  
{  
    Session session = (Session)ctx.getAttribute(FORUM, PageContext.  
SESSION_SCOPE);  
    if ( session==null ) {  
        String driver = ctx.getServletContext().  
getInitParameter("database-driver");  
        String url = ctx.getServletContext().  
getInitParameter("database-url");  
        session = new Session(driver,url);  
        ctx.setAttribute(FORUM,session,PageContext.SESSION_SCOPE);  
    }  
    return session;  
}
```

First, this method checks to see whether a Session object has already been created for this session:

```
Session session = (Session)ctx.getAttribute(FORUM, PageContext.
SESSION_SCOPE);
If a session does not exist, then one is created:
if ( session==null ) {
```

To open a connection to the database, the program must retrieve the database driver and URL. These variables will be used later to open connections to the database as needed. As we mentioned in the previous section, these two values are stored in the web.xml file. The following code retrieves these values:

```
String driver = ctx.getServletContext().
getInitParameter("database-driver");
String url = ctx.getServletContext().
getInitParameter("database-url");
```

These two variables are passed to the constructor of the Session class:

```
session = new Session(driver,url);
ctx.setAttribute(FORUM,session,PageContext.SESSION_SCOPE);
```

The constructor to the Session class is simple. The database driver and URL are passed in and stored. The constructor for the Session class is shown here:

```
/**
 * The constructor. Constructs a Session object
 * from a page context. To create a session
 * object you should use the getSession factory
 * method.
 *
 * @param driver A database driver
 * @param url A database url
 */
Session(String driver,String url)
{
    this.url = url;
    this.driver = driver;
}
```

As the user uses the forum site, database connections are opened and closed as needed. The following method is used to open a connection to the database. This method makes use of the database driver and URL obtained previously:

```
/**
 * Factory method to create sessions. This method
 * will ensure that there is only one object created
 * per user session.
 *
 * @return A Connection object.
 * @exception JspException Thrown on general error.
```

```

*/
Connection getConnection()
throws JspException
{
    try {
        Class.forName(driver);
        Connection conn = DriverManager.getConnection(url);
        return conn;
    } catch ( ClassNotFoundException e ) {
        throw new JspException(e);
    } catch ( SQLException e ) {
        throw new JspException(e);
    }
}

```

Our program can now connect to the database. This is accomplished with the usual JDBC commands for establishing a database connection:

```

Class.forName(driver);
conn = DriverManager.getConnection(url);

```

Now that the database connection has been opened, we can instantiate a new Session object. The object receives the connection to the database and is registered as a session-scoped attribute.

Handling Scope

The forum tag library must frequently create scoped variables to contain the data that the tag library retrieves. Unfortunately, when the scoped variables are created, their scope must be specified using integer constants stored in the PageContext class:

```

/**
 * Called to get the int constant for a character scope
 * value such as "page".
 *
 * @param str A string that should contain "page", "session", etc.
 * @return The numeric form of the specified scope.
 */
static public int getScope(String str)
{
    if ( str==null )
        return PageContext.PAGE_SCOPE;

    if ( str.equalsIgnoreCase("request") )
        return PageContext.REQUEST_SCOPE;
    else if ( str.equalsIgnoreCase("session") )
        return PageContext.SESSION_SCOPE;
    else if ( str.equalsIgnoreCase("application") )
        return PageContext.APPLICATION_SCOPE;
    else return PageContext.PAGE_SCOPE;
}

```

As you can see, the `getScope()` method is simple. The input string is compared against four constant string values that represent each of the four possible scopes. Once the scope is determined, the integer constant for that scope is returned. This method is called any time a scope attribute is specified in any of the forum application tags. This will allow the variable access to occur at the correct scope.

Retrieving a Single Record

One of the primary jobs of the Session class is to perform database queries. All of the JDBC code used by the forum application is concentrated here. The Session class includes various methods for database access. We won't cover each of them in this chapter because they all operate in a similar manner. Instead, we highlight some of database access methods that are representative of how the others work.

One such method is `getUser()`. This method performs one of the most basic operations of the Session class: It retrieves a single record. The `getUser()` method is shown here:

```
/**  
 * Called to load a user from the database.  
 *  
 * @param id The user that is to be loaded.  
 * @return A user object.  
 * @exception JspException Thrown on general error.  
 */  
public User loadUser(String id)  
throws JspException  
{  
    Connection connection = null;  
    PreparedStatement stmt = null;  
  
    try {  
  
        connection = getConnection();  
        stmt = connection.prepareStatement  
            ("select c_uid,c_pwd,c_accesses,c_  
first,c_last,c_bad,c_posted,c_type " +  
                "from t_users " +  
                "where c_uid=?");  
        stmt.setString(1, id.toLowerCase());  
        ResultSet rs = stmt.executeQuery();  
        if ( !rs.next() )  
            return null;  
  
        User user = new User();  
        user.setId(rs.getString("c_uid"));  
        user.setPassword(rs.getString("c_pwd"));  
        user.setAccesses(rs.getInt("c_accesses"));  
        user.setFirst(rs.getDate("c_first"));  
        user.setLast(rs.getDate("c_last"));  
        user.setBad(rs.getInt("c_bad"));  
        user.setPosted(rs.getInt("c_posted"));  
        user.setType(rs.getString("c_type"));  
        return user;  
    } catch (SQLException e) {  
        throw new JspException(e.getMessage());  
    } finally {  
        if (stmt != null)  
            try {  
                stmt.close();  
            } catch (SQLException e) {}  
        if (connection != null)  
            try {  
                connection.close();  
            } catch (SQLException e) {}  
    }  
}
```

```

        } catch ( SQLException e ) {
            throw new JspException(e);
        }

        finally
        {
            try
            {
                if(stmt!=null)
                    stmt.close();
                if(connection!=null)
                    connection.close();
            }
            catch ( SQLException e ) {
                throw new JspException(e);
            }
        }
    }
}

```

The `getUser()` method begins by preparing a SQL statement that will search for a user by that user's user ID. The SQL statement is constructed so that all of the fields associated with that user will be returned in the result set.

The individual values contained in the result set must be copied to the properties of a User object. This is done by a series of calls to the various put methods of the User object.

Not all accesses to the database are as simple as requesting a single record. Often, multiple records will have to be returned, or some update or insertion will need to be performed against the database. Next, we discuss the process used to retrieve a collection of rows from the database. This process is nearly the same as the single-record access method, except that the collection method will need to process records in a loop.

Retrieving a Collection

Often, the data we want to retrieve is more than a single record. Sometimes, the forum application displays a table of data. To display such a table, the program must return a collection of data.

One such table is the list of all of the forum users. This list is accessible from the forum application's administration screens, and allows you to edit or delete users. To display this collection of users, the Session class provides a method named `listUsers()`:

```

/**
 * Called to return a collection of all
 * of the users.
 *
 * @return A collection that contains all of
 * the users.
 * @exception JspException Thrown on general error.
 */

```

```

public Collection listUsers()
throws JspException
{
    Connection connection = null;
    PreparedStatement stmt = null;

    try {
        connection = getConnection();
        Collection c = new LinkedList();
        stmt = connection.prepareStatement
            ("select c_uid,c_pwd,c_accesses,c_
first,c_last,c_bad,c_posted,c_type " +
             "from t_users order by c_uid" );
        ResultSet rs = stmt.executeQuery();
        while ( rs.next() ) {
            User user = new User();
            user.setId(rs.getString("c_uid"));
            user.setPassword(rs.getString("c_pwd"));
            user.setAccesses(rs.getInt("c_accesses"));
            user.setFirst(rs.getDate("c_first"));
            user.setLast(rs.getDate("c_last"));
            user.setBad(rs.getInt("c_bad"));
            user.setPosted(rs.getInt("c_posted"));
            user.setType(rs.getString("c_type"));
            c.add(user);
        }
        return c;
    } catch ( SQLException e ) {
        throw new JspException(e);
    }

    finally
    {
        try
        {
            if(stmt!=null)
                stmt.close();
            if(connection!=null)
                connection.close();
        }
        catch ( SQLException e ) {
            throw new JspException(e);
        }
    }
}

```

This method first instantiates a new ArrayList with an arbitrary length of 3. This collection will hold the individual User objects that will be created. The method begins by submitting a query to the database. This query retrieves all of the users currently registered and orders them by their user ID.

The program then loops through the returned records. It copies each field into the corresponding property of the User class. Once the query has been completely processed, the collection of User objects is returned to the calling method.

Encapsulating Functionality

In addition to retrieving individual and collections of records, another important benefit of moving the database functionality away from the JSP pages is that complex operations can be hidden from the presentation logic contained in the JSP pages. The two examples that we examined so far were simple result sets that were returned in essentially the same form as they were retrieved from the database. Database operations are not always that simple.

One such example from the forum application is what transpires when the user posts a message to one of the forums. When a message is posted to a forum, a new row must be added to the t_messages table. To properly add a message, the program must calculate a new message number. In addition, it must fill in the current date that the message was posted. These tasks are handled by the `createMessage()` method provided by the Session class:

```
/**  
 * Called to create a new forum message.  
 *  
 * @param code The forum that the message is to be created in.  
 * @param from The login id of the user that this message is from.  
 * @param subject The subject of this message.  
 * @param message The actual message text.  
 * @exception JspException Thrown on general error.  
 */  
public void createMessage(String code, String from, String  
subject, String message)  
throws JspException  
{  
    Connection connection = null;  
    PreparedStatement stmt = null;  
  
    try {  
        connection = getConnection();  
        code = code.toUpperCase();  
        stmt = connection.prepareStatement  
                ("select max(c_number)+1 as msgnum  
from t_messages"+  
                 " where c_forum_code = ?");  
        stmt.setString(1, code.toUpperCase());  
        ResultSet rs = stmt.executeQuery();  
        int num;  
        if ( rs.next() )  
            num = rs.getInt("msgnum");  
        else  
            num = 1;  
  
        stmt = connection.prepareStatement  
                ("insert into t_messages(c_forum_code,c_number,c_  
posted,c_subject,c_sender,c_message)+"  
                 "values(?, ?, now(), ?, ?, ?)");  
        stmt.setString(1, code);  
        stmt.setInt(2, num);  
    } catch (SQLException e) {  
        throw new JspException("Error creating message: "+e.getMessage());  
    } finally {  
        if (stmt != null)  
            try {  
                stmt.close();  
            } catch (SQLException e) {}  
        if (connection != null)  
            try {  
                connection.close();  
            } catch (SQLException e) {}  
    }  
}
```

```

        stmt.setString(3, subject);
        stmt.setString(4, from);
        stmt.setString(5, message);
        stmt.execute();

        User user = loadUser(from);
        user.setPosted(user.getPosted() + 1);
        saveUser(user);
    } catch ( SQLException e ) {
        throw new JspException(e);
    }
    finally
    {
        try
        {
            if(stmt!=null)
                stmt.close();
            if(connection!=null)
                connection.close();
        }
        catch ( SQLException e ) {
            throw new JspException(e);
        }
    }
}
}

```

The `createMessage()` method must accomplish two key tasks to post the message. First, it must determine the message number that will be assigned to the newly created message. This is done by doing a SQL SELECT with the max aggregate function. Whatever the maximum message number used by the forum, the new message number will be one higher. Once this number is determined, the message can be posted.

The same Connection object can be reused with the second task. However, a new Statement object is instantiated. The message is inserted into the `t_messages` table using the SQL INSERT INTO command.

The message number, along with the parameter information, is copied to the statement for use by the insert statement. Once the Statement object is prepared, its `execute()` method is called and the new record is created.

Constructing the Tags

In the previous section, we saw how the database queries were constructed to perform three basic types of operations. In this section, we explain how these database operations are tied to the actual tags.

Retrieving a Single Record

First let's construct a tag that will return one record. We've named this tag `<forum:loadUser>`, and it will be responsible for loading an individual user from the database. Listing 11.5 shows the LoadUserTag class file that implements the custom tag.

Listing 11.5 Loading a User (LoadUserTag.java)

```
package com.heaton.forum;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.util.*;
import java.io.*;
import org.apache.taglibs.standard.tag.el.core.*;
import java.sql.*;

/**
 * Example program from Chapter 11
 * JSTL: JSP Standard Template Library Kick Start
 * Published by Sams Publishing
 *
 *
 * This file implements the loadUserTag tag for the
 * forum custom tag library example.
 *
 * @author Jeff Heaton(http://www.jeffheaton.com)
 * @version 1.0
 */

public class LoadUserTag extends TagSupport {
    private String id_el;
    private String var;
    private String scope;
    private String id;

    public void setId(String id) {
        this.id_el = id;
    }

    public void setVar(String var) {
        this.var = var;
    }

    public void setScope(String scope) {
        this.scope = scope;
    }

    /**
     * The start of the tag, used to process the EL
     * expressions.
     *
     * @return A status to indicate if the body is to be
     * processed. Here the body is not processed.
     * @exception javax.servlet.jsp.JspException Thrown on general
     * error.
     */
    public int doStartTag()
```

```

throws javax.servlet.jsp.JspException
{
    id = (String)ExpressionUtil.evalNotNull(
        "setUser", "id", id_el, String.class, this, pageContext);

    return SKIP_BODY;
}

/**
 * The ending tag method is called when the tag
 * is ready to complete. It is here that
 * the user is loaded.
 *
 * @return A status code that indicates if the rest of the
 * page is to be processed.
 * @exception JspException Thrown on general error.
 */
public int doEndTag() throws JspException {
    Session session = Session.getSession(pageContext);
    User user = session.loadUser(id);
    if ( user==null ) {
        pageContext.removeAttribute(var);
    } else {
        pageContext.setAttribute(var,user);
    }

    return EVAL_PAGE;
}
}

```

You'll notice that several methods and properties are used to implement the loadUser tag. Let's examine these components so you can easily create your own tag libraries.

Accessing Attributes

One of the most basic components of any JSP tag is the attributes given to that tag. The `<forum:loadUser>` tag must be provided with the user ID to be loaded. Likewise, the `<forum:loadUser>` tag must be provided with the name of the scoped variable in which the user is to be stored. Finally, the `<forum:loadUser>` tag must be provided with the scope that the scoped variable must be created in.

This gives the `<forum:loadUser>` three attributes, named id, var, and scope. Calling `<forum:loadUser>` to load the admin user might look like this:

```
<forum:loadUser id="admin" var="user" scope="page"/>
```

Your program must be able to read in each of these values if the tag is to function properly. Reading in these three values would be quite easy, since they are just simple constant values. There is no command that actually reads in these values. To gain access to these values, you must create set methods for each of them. The Web server calls your

set methods and allows your tag to receive all three attributes. For example, we use the `setId()` method to retrieve the value of the id attribute:

```
public void setId(String id) {  
    this.id_el = id;  
}
```

This set tag stores the value specified for the id attribute in the variable `id_el`. If your tag is provided only to support simple constant values, this is sufficient.

Evaluating EL Expressions

If you would like your tag to be able to accept EL expressions, you must take a few additional steps. Consider the following tag, which uses an EL expression:

```
<forum:loadUser id="${param.uid}" var="user" scope="page"/>
```

In this tag, the user ID is not just a simple constant value. When our set method is called, it is passed the string `${param.uid}`. We want this value to be evaluated, but it is not evaluated in the set method. Instead, the set method simply stores it away in the `id_el` variable for the `doStartTag()` method to evaluate. The only operation performed by the `doStartTag()` method of the `<forum:loadUser>` tag is to evaluate the value of the id attribute:

```
id = (String)ExpressionUtil.evalNotNull(  
    "setUser", "id", id_el, String.class, this, pageContext);
```

This code allows you to call the Apache expression evaluator to evaluate the EL expression.

The `ExpressionUtil.evalNotNull()` method is provided by the Apache implementation of JSTL and allows you to access individual attributes of your tags. Its primary benefit is that it is capable of evaluating EL expressions.

WARNING

The `ExpressionUtil` class is not a part of the standard JSTL implementation. This is a class that the Apache implementation of JSTL included for convenience. The command to evaluate an EL expression when using an implementation of JSTL other than Apache may be slightly different. It is also important that the JAR files `servlet.jar`, `standard.jar`, and `jstl.jar` all be added to the classpath so that your tag has access to the `ExpressionUtil` class while your tag is being compiled. This is covered in greater detail in [Appendix B](#), "Installing Tomcat and JSTL."

Once the result of the expression has been stored in the `id` variable, the work of the `doStartTag()` method is done. The method can now return with the following statement:

```
return SKIP_BODY;
```

Because the `<forum:loadUser>` tag does not expect body content, we can safely return the `SKIP_BODY` identifier to halt further evaluation of the tag's body. After the body is processed, the `doEndTag()` method is called.

The doEndTag() Method

Most bodyless tags perform their primary action in the `doEndTag()` method. In the case of the `<forum:loadUser>` tag, the `doEndTag()` method begins by obtaining the Session object so that a database command can be invoked:

```
Session session = Session.getSession(pageContext);
```

Once we have access to the session, we can load the user. This is done by calling the `loadUser()` method of the Session object:

```
User user = session.loadUser(id);
```

We discussed the `loadUser()` method earlier in this chapter. If the user is found, the `loadUser()` method returns null. If the value returned is null, then we want to remove the scoped variable, specified by the `var` attribute, from the context. This is done with the following code:

```
if ( user==null ) {
    pageContext.removeAttribute(var);
} else {
```

If a valid user is returned, then we store this user in the scoped variable specified by the `var` attribute. The following code sets the appropriate scoped variable with the User object:

```
    pageContext.setAttribute(var,user);
}
```

Once the User object has been copied, the tag's work is nearly done. Now, the tag must return with a constant that specifies the next action to be taken:

```
return EVAL_PAGE;
```

In this example, we use the constant `EVAL_PAGE`, which allows the rest of the page to be evaluated. If the `SKIP_PAGE` constant were returned, the remaining data on the page would not be evaluated.

Retrieving a Collection

Collections are an important part of the JSTL tag library. Many tags are provided that allow you to work with collections. This includes not only accessing the data members of the collections, but also iterating through them. By having your tag return a collection, you are able to use the `<c:forEach>` tag to process the elements of your collection.

One such example of this is the `<forum:listUsers>` tag. This tag returns a complete list of the users as a collection. This allows admin screens to display a complete listing of users for editing. The `<forum:listUsers>` tag is implemented inside `ListUsersTag.java`. The source code to `ListUsersTag.java` is shown in Listing 11.6.

Listing 11.6 The listUsers Tag (ListUsersTag.java)

```
package com.heaton.forum;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.util.*;
import java.io.*;
import org.apache.taglibs.standard.tag.el.core.*;
import java.sql.*;

/**
 * Example program from Chapter 11
 * JSTL: JSP Standard Template Library
 * Published by Sams Publishing
 *
 *
 * This file implements the listUsers tag for the
 * forum custom tag library example.
 *
 * @author Jeff Heaton(http://www.jeffheaton.com)
 * @version 1.0
 */

public class ListUsersTag extends TagSupport {
    private String var;
    private String scope;

    public void setVar(String var) {
        this.var = var;
    }

    public void setScope(String scope) {
        this.scope = scope;
    }
    /**
     * The start of the tag, used to process the EL
     * expressions.
     *
     * @return A status to indicate if the body is to be
     * processed. Here the body is not processed.
     * @exception javax.servlet.jsp.JspException Thrown on general
     * error.
     */
    public int doStartTag()
```

```

throws javax.servlet.jsp.JspException
{
    return SKIP_BODY;
}

/**
 * The ending tag method is called when the tag
 * is ready to complete. It is here that
 * the users are listed.
 *
 * @return A status code that indicates if the rest of the
 * page is to be processed.
 * @exception JspException Thrown on general error.
 */
public int doEndTag() throws JspException {
    Session session = Session.getSession(pageContext);
    Collection list = session.listUsers();
    pageContext.setAttribute(var,list,Session.getScope(scope));
    return EVAL_PAGE;
}
}

```

The `<forum:listUsers>` tag is similar in structure to the `<forum:loadUser>` tag. The primary difference is that the tag returns a collection rather than an individual user record.

Encapsulating Functionality

So far, we have seen tags that simply map to database queries. While a program will usually have many such tags, another key feature of tag libraries is their ability to encapsulate common functionality. `PostMessageTag.java` is shown in Listing 11.7.

Listing 11.7 Posting a Message (`PostMessageTag.java`)

```

package com.heaton.forum;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.util.*;
import java.io.*;
import org.apache.taglibs.standard.tag.el.core.*;
import java.sql.*;

/**
 * Example program from Chapter 11
 * JSTL: JSP Standard Template Library
 * Published by Sams Publishing
 *
 *
 * This file implements the postMessages tag for the
 * forum custom tag library example.
 *
 * @author Jeff Heaton(http://www.jeffheaton.com)
 * @version 1.0
 */

```

```

public class PostMessageTag extends TagSupport {
    private String code_el;
    private String code;
    private String from_el;
    private String from;
    private String subject_el;
    private String subject;
    private String message_el;
    private String message;

    public void setCode(String code) {
        this.code_el = code;
    }

    public void setFrom(String from) {
        this.from_el = from;
    }

    public void setSubject(String subject) {
        this.subject_el = subject;
    }

    public void setMessage(String message) {
        this.message_el = message;
    }

    /**
     * The start of the tag, used to process the EL
     * expressions.
     *
     * @return A status to indicate if the body is to be
     * processed. Here the body is not processed.
     * @exception javax.servlet.jsp.JspException Thrown on general
     * error.
     */
    public int doStartTag()
        throws javax.servlet.jsp.JspException
    {
        code = (String)ExpressionUtil.evalNotNull(
            "postMessage", "code", code_el, String.class, this,
            pageContext);

        from = (String)ExpressionUtil.evalNotNull(
            "postMessage", "from", from_el, String.class, this,
            pageContext);

        subject = (String)ExpressionUtil.evalNotNull(
            "postMessage", "subject", subject_el, String.class, this,
            pageContext);

        message = (String)ExpressionUtil.evalNotNull(
            "postMessage", "message", message_el, String.class, this,
            pageContext);
        return SKIP_BODY;
    }
}

```

```

/**
 * The ending tag method is called when the tag
 * is ready to complete. It is here that
 * the message is posted.
 *
 * @return A status code that indicates if the rest of the
 * page is to be processed.
 * @exception JspException Thrown on general error.
 */
public int doEndTag() throws JspException {
    Session session = Session.getSession(pageContext);
    session.createMessage(code, from, subject, message);
    return EVAL_PAGE;
}
}

```

From the tag's perspective, this is an easy operation. All of the work is done by the `createMessage()` method of the Session object. We defined this method earlier in this chapter.

The `doEndTag()` method begins by establishing a Session object. With access to the session, the method calls the `createMessage()` method. Nothing is returned by this method; the tag simply returns with the directive to process the rest of the page.

Tags with Bodies

Sometimes, you will want to create a custom tag that also has a body. One example of this is the `<forum:expire>` tag. This tag works like an if statement. If the user's session has expired, the program executes the body of the `<forum:expire>` tag. This tag is used in instances such as the following:

```

<forum:expire>
    <c:redirect url="index.jsp" />
</forum:expire>

```

This tag checks to see whether the user's session has expired. If it has, then the program executes the body of the tag. The body contains a single JSTL command that redirects the page back to the `index.jsp` file.

Let's now look at how to construct a tag that also contains a body. The source code behind the `<forum:expire>` tag is shown in Listing 11.8.

Listing 11.8 Checking to See if the User Has Expired (ExpireTag.java)

```

package com.heaton.forum;
import javax.servlet.jsp.JspTagException;
import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.jstl.core.ConditionalTagSupport;

/**

```

```

* Example program from Chapter 11
* JSTL: JSP Standard Template Library Kick Start
* Published by Sams Publishing
*
*
* This file implements the expireTag tag for the
* forum custom tag library example. This tag is
* used to determine if the session has timed out.
*
* @author Jeff Heaton(http://www.jeffheaton.com)
* @version 1.0
*/
}

public class ExpireTag extends ConditionalTagSupport {
    /**
     * @return true if the Session object has not been created in the
     * session scope.
     */
    protected boolean condition() throws JspTagException {
        return pageContext.getAttribute(Session.FORUM, PageContext.
SESSION_SCOPE) == null;
    }
}

```

With a tag such as `<forum:expire>`, the majority of processing is done in the `condition()` method rather than the `doEndTag()` method. This is because we overrode the `ConditionalTagSupport` class. The `ConditionalTagSupport` class is used to provide tags similar to `<c:if>`. The method simply returns true if the body is to be executed or false if it is not.

There are other helper classes like `ConditionalTagSupport`. You are not required to use these helper classes. For example, we could have implemented the `expire` tag using the `TagSupport` class. Other helper classes provided by JSTL include:

- `LoopTagStatus`— Provides status information for loops
- `LoopTagSupport`— Provides tags that repeat their body statements in a loop
- `LocaleSupport`— Used for I18N support
- `SQLExecutionTag`— Provides tags that use SQL
- `ResultSupport`— Provides your own result sets

The tags that we've examined are ones you'll use in most cases. For more information about these classes, refer to the JavaDoc pages at <http://java.sun.com/>.

Creating the JAR File

Tag libraries are usually distributed as JAR files. In this section, we show you how to compile the `forum` application's JAR file. We provide a script file that allows you to compile the JAR file. If you are using Unix, you should use the `build_taglib.sh` script. If you are using Windows, you should use `build_taglib.bat`. The contents of both script files are essentially the same. The `build_taglib.sh` script file is shown in Listing 11.9.

Listing 11.9 Building the Taglib (build_taglib.sh)

```
javac ./com/heaton/forum/LoadUserTag.java
javac ./com/heaton/forum/SaveUserTag.java
javac ./com/heaton/forum/LoginUserTag.java
javac ./com/heaton/forum/ExpireTag.java
javac ./com/heaton/forum/NewUserTag.java
javac ./com/heaton/forum/ListForumsTag.java
javac ./com/heaton/forum/ListMessagesTag.java
javac ./com/heaton/forum/LoadForumTag.java
javac ./com/heaton/forum/PostMessageTag.java
javac ./com/heaton/forum/DeleteMessageTag.java
javac ./com/heaton/forum/DeleteUserTag.java
javac ./com/heaton/forum>ListUsersTag.java
javac ./com/heaton/forum/EditForumTag.java
javac ./com/heaton/forum/EditUserTag.java
javac ./com/heaton/forum/CreateForumTag.java
javac ./com/heaton/forum/DeleteForumTag.java
jar cvf forum.jar ./com/heaton/forum/*.class
```

The script file in Listing 11.9 executes the Java compiler on each of the tag files. Once the tag files are compiled, the jar command is used to package all the class files into a single JAR file. You need to place this JAR file, named forum.jar, in the `/lib` directory of your Web server. The examples directory that you will download from the Sams Publishing Web site will already contain the JAR file in the `/lib` directory.

WARNING

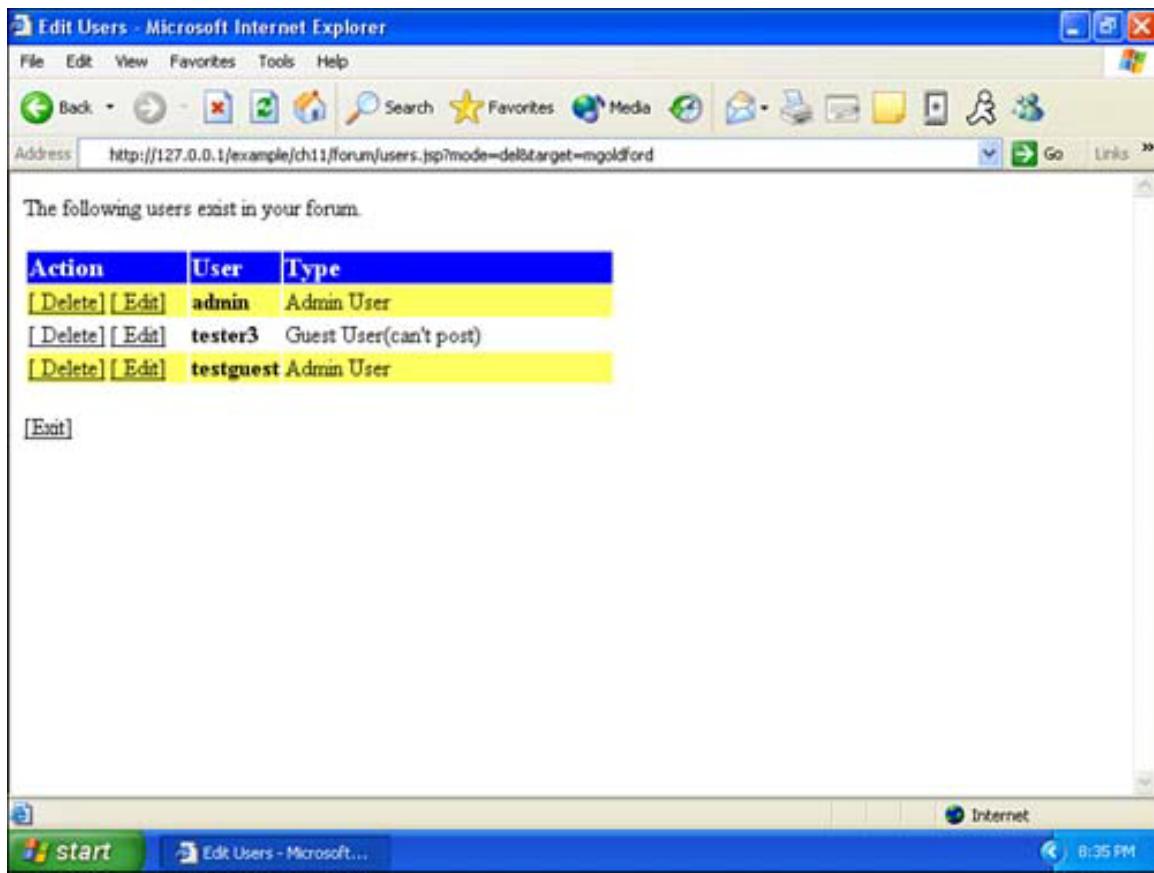
If you get a compile error, you must make sure that your classpath has all of the required JAR files.

You have now learned to create tags. This includes setting up the configuration files as well as compiling the actual Java source files. Now, let's see how to use these tags.

Using the Tags from JSP

Tags are used from JSP pages. Let's look at a JSP page—one of the admin pages—that uses some of the tags we just created. The user administration page must present a list of all the users of the system. This page, shown in Figure 11.1, uses several of the forum application's tags.

Figure 11.1. Listing the users.



The source code to the user administration JSP page is shown in [Listing 11.10](#).

Listing 11.10 Our User Admin Page (users.jsp)

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt" %>
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jstl/core-rt" prefix="c-rt" %>
<%@ taglib uri="http://java.jeffheaton.com/taglib/jstl/forum"
   prefix="forum" %>

<forum:expire>
  <c:redirect url="index.jsp" />
</forum:expire>

<c:if test="${user.type!='A'}">
  <c:redirect url="main.jsp" />
</c:if>

<c:if test="${param.mode=='del'}">
  <forum:deleteUser id="${param.target}" />
</c:if>

<html>
  <head>
    <title><fmt:message key="editusers.title" bundle="${lang}" />
```

```

</title>
</head>

<body link="#000000" vlink="#000000" alink="#000000">
    <p><fmt:message key="editusers.ins" bundle="${lang}"/></p>
    <forum:listUsers var="list"/>

    <table border="0" width="410">
        <tr>
            <td bgcolor="#0000FF" width="115">
                <b>
                    <font color="#FFFFFF" size="4">
                        <fmt:message key="word.action" bundle="${lang}"/></font>
                </b>
            </td>

            <td bgcolor="#0000FF" width="45">
                <b>
                    <font color="#FFFFFF" size="4">
                        <fmt:message key="word.user" bundle="${lang}"/></font>
                </b>
            </td>

            <td bgcolor="#0000FF" width="246">
                <b>
                    <font color="#FFFFFF" size="4">
                        <fmt:message key="word.type" bundle="${lang}"/></font>
                </b>
            </td>
        </tr>
    </table>

    <c:forEach var="row" items="${list}" varStatus="status">
        <c:url value="${pageContext.request.requestURI}" var="del">
            <c:param name="mode" value="del" />

            <c:param name="target" value="${row.id}" />
        </c:url>

        <c:url value="edituser.jsp" var="edit">
            <c:param name="mode" value="edit" />

            <c:param name="target" value="${row.id}" />
        </c:url>

        <jsp:useBean id="status"
            type="javax.servlet.jsp.jstl.core.LoopTagStatus" />

        <c-rt:choose>
            <c-rt:when test="<%=>status.getCount()%2==0%">
                <c:set var="color" value="#FFFFFF" />
            </c-rt:when>

            <c-rt:otherwise>
                <c:set var="color" value="#FFFF66" />
            </c-rt:otherwise>
        </c-rt:choose>
    </c:forEach>

```

```

<tr bgcolor=<c:out value="${color}" />>
<td width="115">
    <a href=<c:out value="${del}" />>[<fmt:message key="word.delete" bundle="${lang}" />]</a>
    <a href=<c:out value="${edit}" />>[<fmt:message key="word.edit" bundle="${lang}" />]</a>
</td>

<td width="35">
    <b>
        <c:out value="${row.id}" />
    </b>
</td>

<td width="246">
    <c:choose>
        <c:when test="${row.type=='A'}"><fmt:message key="type.a" bundle="${lang}" /></c:when>
        <c:when test="${row.type=='R'}"><fmt:message key="type.r" bundle="${lang}" /></c:when>
        <c:when test="${row.type=='G'}"><fmt:message key="type.g" bundle="${lang}" /></c:when>
        <c:otherwise>
            <c:out value="${row.type}" />
        </c:otherwise>
    </c:choose>
</td>
</tr>
</c:forEach>
</table>

<br />

<a href="admin.jsp">[<fmt:message key="word.exit" bundle="${lang}" />]</a>
</body>
</html>

```

First, the JSP page must use the appropriate import tags to gain access to the forum tag library. At the top of the JSP page, among the other tags, you can see the import tag used to import the forum tag library:

```
<%@ taglib uri="http://java.jeffheaton.com/taglib/jstl/forum"
prefix="forum" %>
```

As you can see, the taglib statement uses the URI <http://java.jeffheaton.com/taglib/jstl/forum>. This is the same URI specified earlier when we created the tag library. This tag makes the forum application's tag library

accessible under the prefix `forum`. Now that the `forum` tag library is accessible, we can begin to use it.

The first thing the user administration page does is ensure that the user's session is still valid. If the user session is invalid, this means that the user has been inactive for too long and has been logged off:

```
<forum:expire>
  <c:redirect url="index.jsp" />
</forum:expire>
```

Next, the page must check to see whether the user is actually an administrator. If the user is not an administrator, the user should not have made it to this page, because the page is accessible only from the main administration screen. However, it is still important to actually check to see whether the user has administrative access. This is because it is possible that the user might have directly entered the URL to this page, thus bypassing previous security. By doing this check, we make absolutely sure that this page has been entered by legitimate means and by an administrative user. This check is done using the user session variable:

```
<c:if test="${user.type != 'A'}">
  <c:redirect url="main.jsp" />
</c:if>
```

At this point, we have successfully validated that the user's session still exists and that the user is a legitimate administrative user. We are now ready to perform the actual duty of this page. As previously stated, the primary purpose of this page is to display every user's login name with links that allow the administrator to delete or edit users.

To list all the users, we must first obtain a collection of every user record contained in the `t_users` table. We do this by using the `<forum:listUsers>` tag. This tag is called specifying a scoped variable in which we want to place the collection:

```
<forum:listUsers var="list"/>
```

Now that we have access to the collection of users, we can iterate through them. We do this by using the familiar `<c:forEach>` tag:

```
<c:forEach var="row" items="${list}"
varStatus="status">
```

We must display the user's ID from each of these row variables:

```
<td width="35">
  <b>
    <c:out value="${row.id}" />
  </b>
</td>
```

As the user collection is iterated through, each user's record allows the user's ID to be displayed. In addition, we construct valid URLs for both the edit and delete links.

By using our own tags, we have now isolated the database access to our tag library. Rather than using the `<sql:query>` tag to access the database, we use our own tag that hides the exact table and column names.

Summary

In this chapter, we showed you how to create your own tag libraries. We described the steps necessary to allow your tags to process EL-based expressions. It is important to design your tag libraries to be compatible with the JSTL tag libraries. That way, you won't have to design the common formatting and iteration tags that make up the JSTL tag library.

In the next (and final) chapter, you'll learn how to tie everything together. We explain how to deploy Web applications that consist of a mixture of custom tag libraries, JSTL, and JSP code. These components can all be combined into one common file called a Web Archive (WAR) file.

Debugging is an important aspect of Web development. In the next chapter, we also examine how to debug JSTL tag libraries. We show you how to output trace information, read error messages, and use the JDK 1.4 logging API with Tomcat.

Chapter 12. Debugging and Deploying Tag Libraries

Up to this point, you have learned how to use the JSTL tag libraries and how to create your own tag libraries. There are two other important factors that you must also consider when developing a Web application.

Debugging is always a considerable part of the development process. Programs rarely work as designed on the first try. In this chapter, we show you how to read error messages generated by the Web server. You will also learn how to use an integrated development environment (IDE) to debug your Web applications.

Packaging and deployment of your Web applications is another important factor. Web applications are often deployed onto multiple computers. Properly packaging your application makes this deployment process easy. In this chapter, we also show you how to deploy your Web applications.

Debugging

Debugging is a critical process for any programming endeavor. JSTL, like any other programming language, is susceptible to programming errors. To effectively program JSTL and custom tag libraries, you must know how to combat these errors. In this section, we show you debugging techniques you can use to debug errors you're likely to encounter when using JSTL and your own custom tag libraries.

Various types of errors can be returned from a JSTL page. The error reports that JSTL displays are designed to provide information that will lead you to the cause of the problem.

Error reports returned from a JSP server may look like a complex jumble of text and numbers, but you must learn how to read them to determine the cause of the error. Several types of errors are common with JSTL pages; they range from general exceptions to JSTL-specific parsing errors. We begin by examining a general exception.

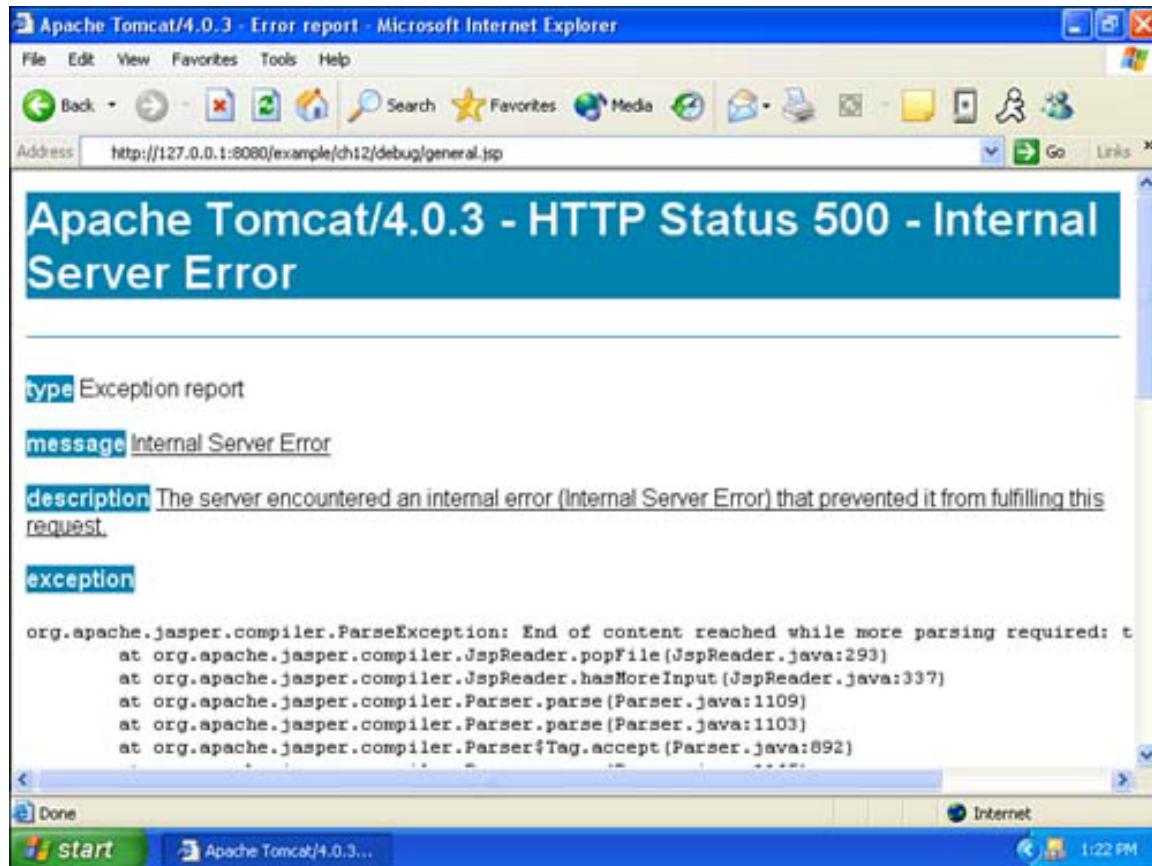
General Exceptions

An application displays a general exception whenever it executes a JSP page that causes a Java exception to be thrown. More than one type of exception can be thrown to a JSP page. Any exception that can be thrown by Java can cause a JSP page to terminate if the exception is not caught.

General exceptions have many causes, and we won't attempt to list them all. In this section, we examine the exception that is thrown when the JSTL processor encounters

beginning and ending tags that do not match. This can often happen when you forget an ending tag. When such an error occurs, you'll see output similar to [Figure 12.1](#).

Figure 12.1. A general JSTL exception.



In [Figure 12.1](#), we neglected to include an ending tag, as the exception indicates. The JSP page that causes the error is shown in [Listing 12.1](#).

Listing 12.1 A Missing Ending Tag (general.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
    <head>
        <title>General JSTL</title>
    </head>

    <body>
        <c:forEach var="i" begin="1" end="10" step="1">

    </body>
</html>
```

A missing ending tag error occurs as a result of an exception being thrown by the parser. With this type of error, you are not provided with the line number within the JSP page.

where this error occurred. Instead, you are given a line number in the compiled servlet. While this number is not as helpful, the error text does yield some information:

```
org.apache.jasper.compiler.ParseException: End of content reached  
while more parsing required: tag nesting error?
```

As you can see, Tomcat has reported that an ending tag is missing.

WARNING

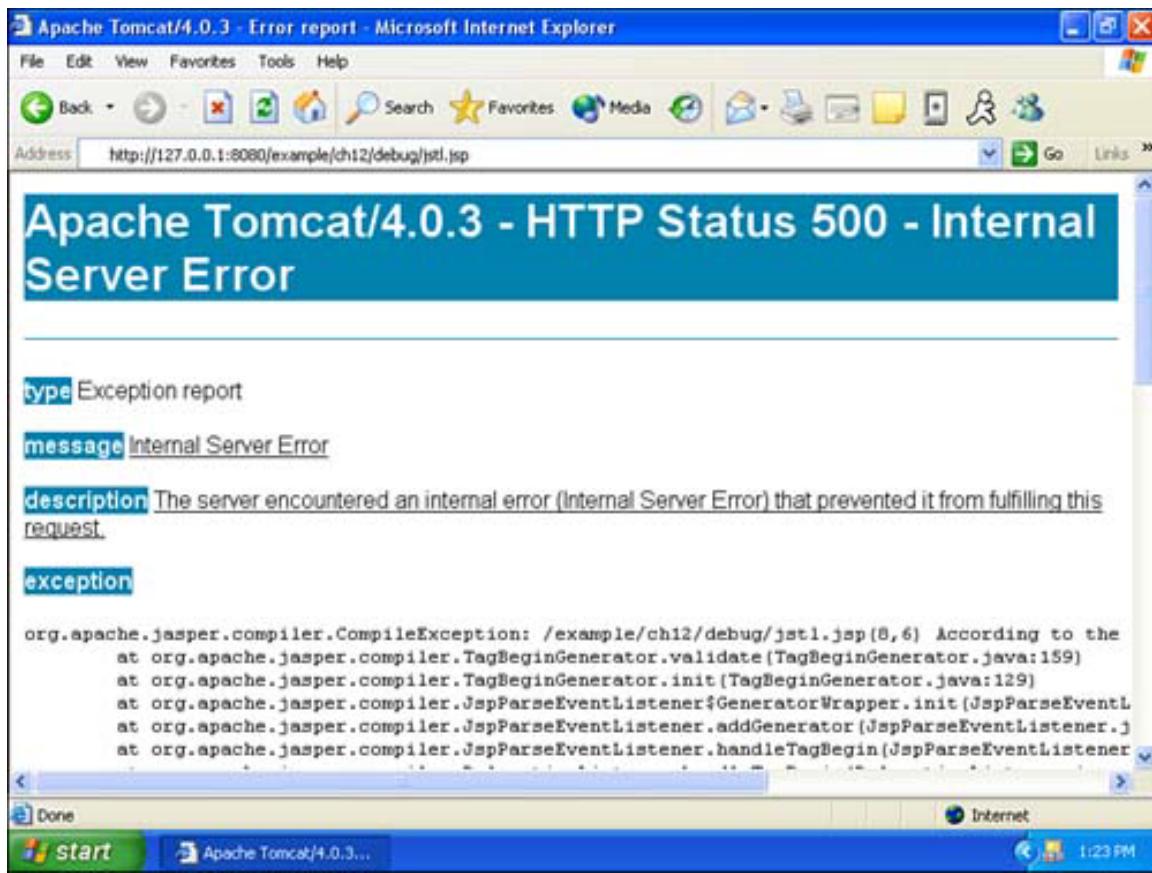
Note that these errors are tied not only to which Web server you use but also to the version. These errors may look different on different Web servers and versions of those Web servers.

In the next section, we examine a JSTL-specific error that gives you the line number and column that caused the error.

A JSTL-Specific Error

A JSTL-specific error can occur when you forget an attribute to JSTL tag. There are other causes, but forgetting required attributes is one of the most common. When you receive a JSTL-specific exception, the output looks similar to that shown in [Figure 12.2](#).

Figure 12.2. A JSTL-specific exception.



A JSTL-specific error gives the line number and column that this error occurred on. This allows you to determine the exact part of the JSP page that caused this error. If you look closely at the first line of the exception thrown, you'll notice the following text:

```
/example/ch12/debug/jstl.jsp(8,6) According to the TLD attribute
    value is mandatory for tag out
```

This text tells you exactly which file caused the error. In this case, the error was caused by a file named jstl.jsp. The numbers following the filename tell you exactly which line and column in the JSP page caused this error. The number 8 indicates the eighth line, and the number 6 indicates the sixth column. Using this information you can quickly track down exactly which tag caused the error that JSTL is complaining about.

For this error, we have simply forgotten to include the value attribute for the `<c:out>` tag. To see this program, refer to [Listing 12.2](#).

Listing 12.2 A Missing Attribute (jstl.jsp)

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
    <head>
        <title>Count to 10 Example (using JSTL)</title>
    </head>
```

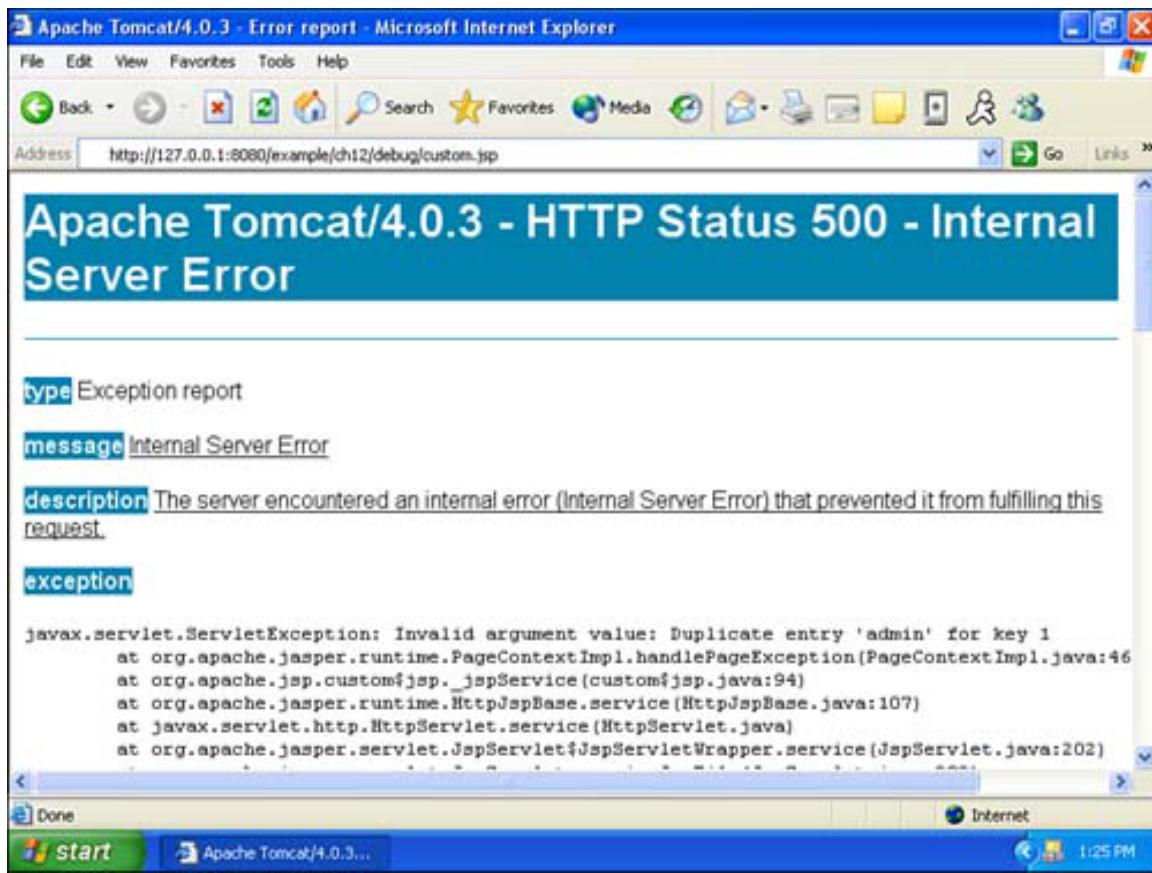
```
<body>
  <c:forEach var="i" begin="1" end="10" step="1">
    <c:out />
    <br />
  </c:forEach>
</body>
</html>
```

The errors that we've examined are the result of JSTL code. Let's now look at how to handle errors that occur inside your own tag libraries.

An Error in a Tag Library

In [Chapter 11](#), we showed you how to create your own custom tag libraries. These custom tag libraries are Java programs, and like all Java programs, they can develop errors. If an error occurs in your custom tag, it is likely that a general exception will be thrown. You must recognize this form of general exception so that you can quickly determine which of your tags has caused the error. If you receive an error from one of your custom tag libraries, the output from your browser will look similar to that shown in [Figure 12.3](#).

Figure 12.3. A tag library exception.



As you can see in [Figure 12.3](#), an error is being raised by the custom tag library. To understand the source of this error, we must examine the JSP code, shown in [Listing 12.3](#), that produced the error.

Listing 12.3 A Custom Tag Error (custom.jsp)

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

<html>
  <head>
    <title>Custom Tag Error</title>
  </head>

  <body>
    <forum:newUser id="admin" password="admin" var="user"
      scope="session"/>
  </body>
</html>
```

The cause of our error is that we are attempting to create a user that has the same name as an existing user. We are attempting to create a user named admin, and the forum application should already have a user named admin. Therefore, this tag throws an error.

Most bugs are not as simple as the ones we've discussed. Often, your code will have logic errors and will not throw an exception, yet the code will not do what it is supposed to do, either. To handle this sort of bug, you must use a debugger. Debuggers are usually made available as part of an IDE. In the next section, we step you through the code of our custom tag library using a debugger.

Debugging with an IDE

An integrated development environment (IDE) is a GUI application that allows you to develop a Java application using a convenient editor. Though using an IDE makes it convenient to develop your application, one of the chief benefits to using an IDE is debugging. You have likely already used an IDE to debug your Java applications. It is also possible to use an IDE to debug a custom tag library, such as the tag library used in [Chapter 11](#).

In this section, we show you how to use an IDE to debug a custom tag library. Using the IDE, you can easily debug your application as you step through your routines and observe the values of variables as your program runs. This can be invaluable for debugging your program and learning why it does what it does.

Commercial IDEs include Visual Cafe, JBuilder, and Visual Age. Many IDEs are available for free. For example, Sun's Forte includes a community edition that you can download for free [Forte at <http://java.sun.com/>](http://java.sun.com/).

In addition, the Eclipse IDE is a completely open source project being developed by IBM. Next, we show you how to install this IDE to use with Tomcat.

Introducing Eclipse

Eclipse is an advanced and very popular IDE for the Java platform. There is some speculation as to where the name for Eclipse came from. Some say that it IBM meant to imply that they will "eclipse" Sun Microsystems in the Java tools market. Whatever the meaning behind the name, it is a valuable tool.

You can download Eclipse from the IBM Web site at <http://www.eclipse.org/>. The Eclipse IDE can be used with many platforms. If you're using Windows or Linux, you can download a binary version that is ready for install.

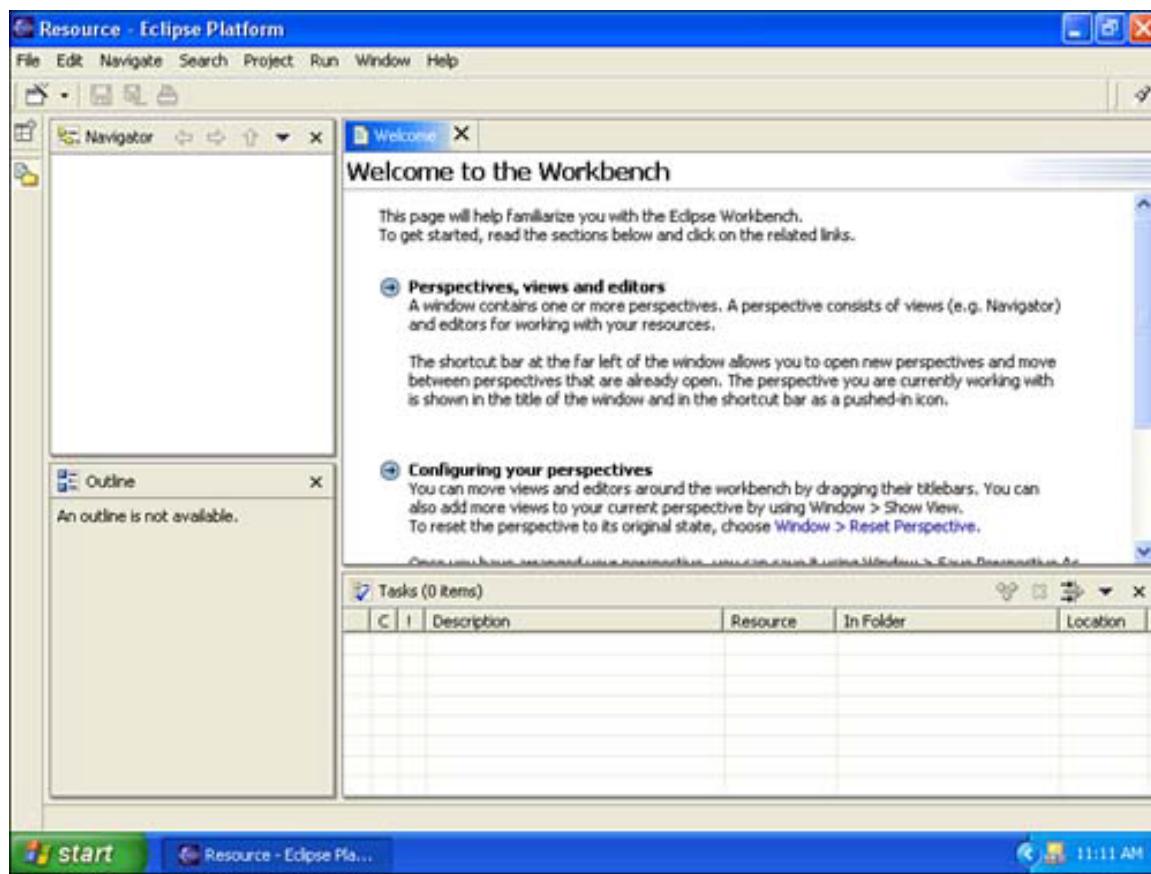
When you download Eclipse, you'll notice that the program is in one large Zip archive. Inside this archive is a single folder, named `eclipse`. You should copy the `eclipse` folder to a convenient location on your computer. Inside the `eclipse` archive is a single executable file named `eclipse.exe`. Double-click this file to start the Eclipse system.

It may be convenient to create a shortcut, or link, to this executable and place this shortcut at a convenient location, such as your desktop.

The Eclipse IDE does not come with a JDK. You must have already downloaded and installed a JDK prior to installing Eclipse. We suggest you use JDK version 1.4 or higher. Once you have installed the JDK and Eclipse on your system, you are ready to begin using Eclipse.

When Eclipse first starts, you'll see the blue Eclipse splash screen as the program loads. Once Eclipse loads, you'll see the Eclipse main screen, shown in [Figure 12.4](#).

Figure 12.4. The Eclipse IDE.



For the IDE debugging example shown in this chapter, we debug the tag library we presented in [Chapter 11](#). To debug the forum application, you must set up a project file that builds the forum tag library and includes the Tomcat Web server. In the next section, you'll learn how to set up a project that uses Tomcat.

Using Eclipse with Tomcat

Debugging a Web application is not as simple as debugging a normal Java application using an IDE. To debug a JSP-based application, you must bring the entire Web server, in

this case Tomcat, into the debugging context. There are several ways that you can do this. One approach is to use a third-party plug-in that has been developed for the Eclipse IDE.

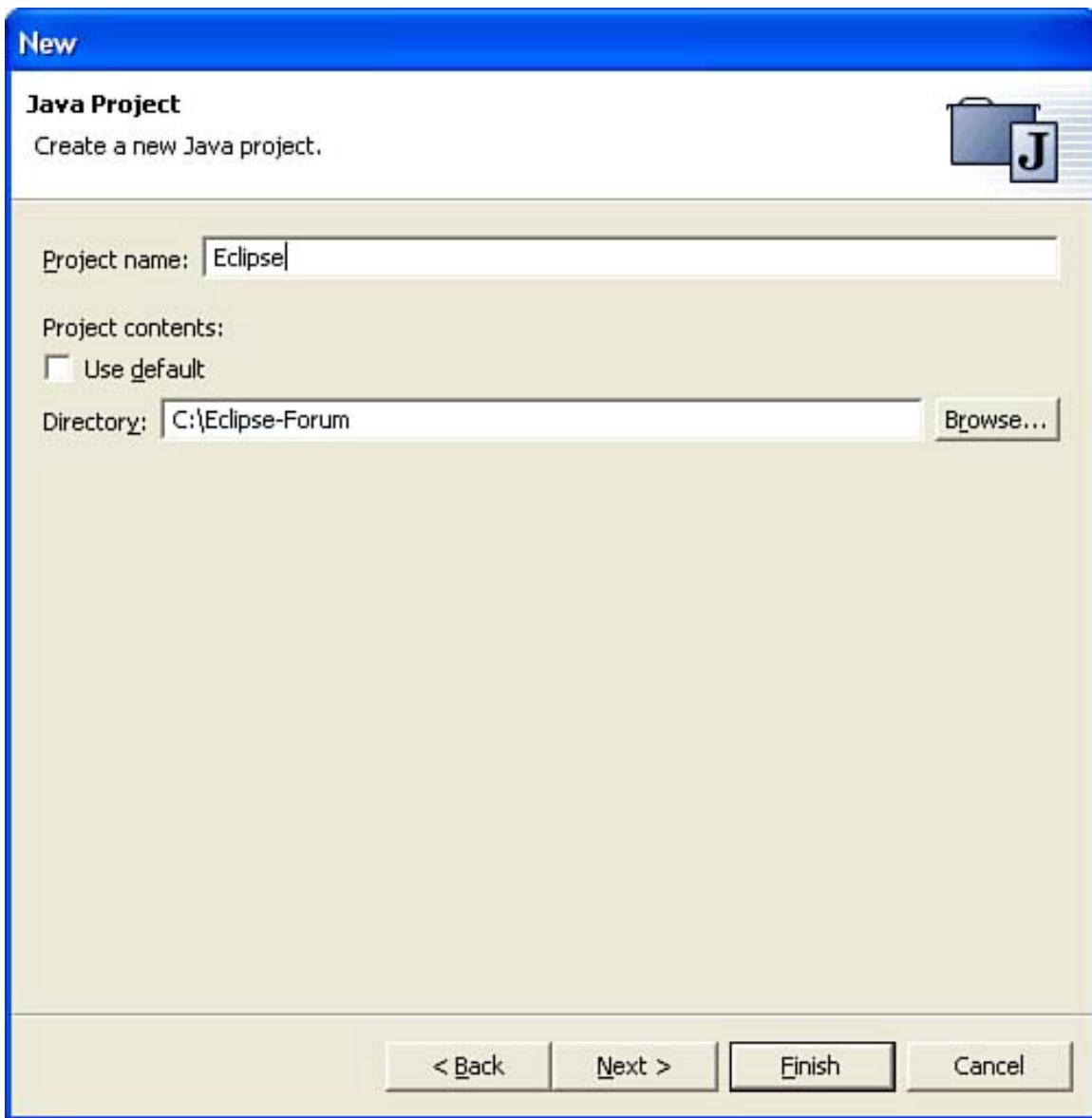
Sysdeo developed the Tomcat plug-in that allows you to use Tomcat with Eclipse. This free tool makes it easier to work with Tomcat and Eclipse. You can download this tool from <http://www.sysdeo.com/eclipse/tomcatPlugin.html>.

For our examples, we don't use a third-party Tomcat plug-in. It is easy enough to configure Eclipse to recognize the JAR files included with the Tomcat distribution. However, if the process we describe here causes difficulty, you might consider using a third-party plug-in for Tomcat.

To debug your Web application with Eclipse, you must set up an Eclipse *project*. Eclipse stores projects in individual folders. These folders contain the settings, Java files, and other information needed to execute to that project. You can download the project file being used here from the Sams Publishing Web site. This folder is named `eclipse-forum`; copy the folder to a convenient location on your computer. The source files available from the Web site contain a sample project in a directory named `eclipse-forum`. In this chapter, we'll assume you've copied the folder to your local directory (`c:\eclipse-forum`).

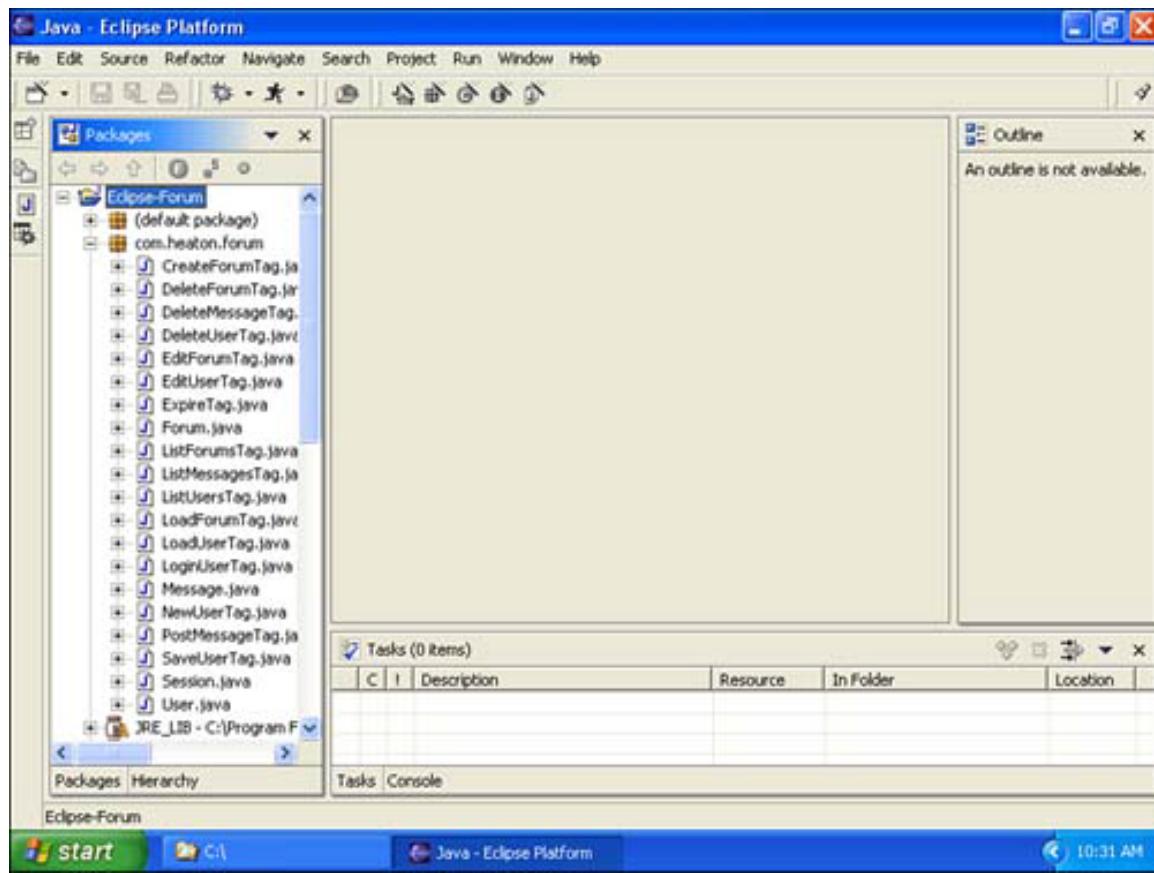
At this point, you must inform Eclipse of the folder's existence by creating a new project. When you choose to create a new project in Eclipse, you'll see a screen similar to the one shown in [Figure 12.5](#).

Figure 12.5. Creating a project.



We are not really creating a new project, but it is necessary to use this method to make Eclipse aware of the forum folder. Simply type in the name of the copied folder, and Eclipse will begin using that folder. Once you've successfully created a new project, you'll see a screen like the one shown in [Figure 12.6](#).

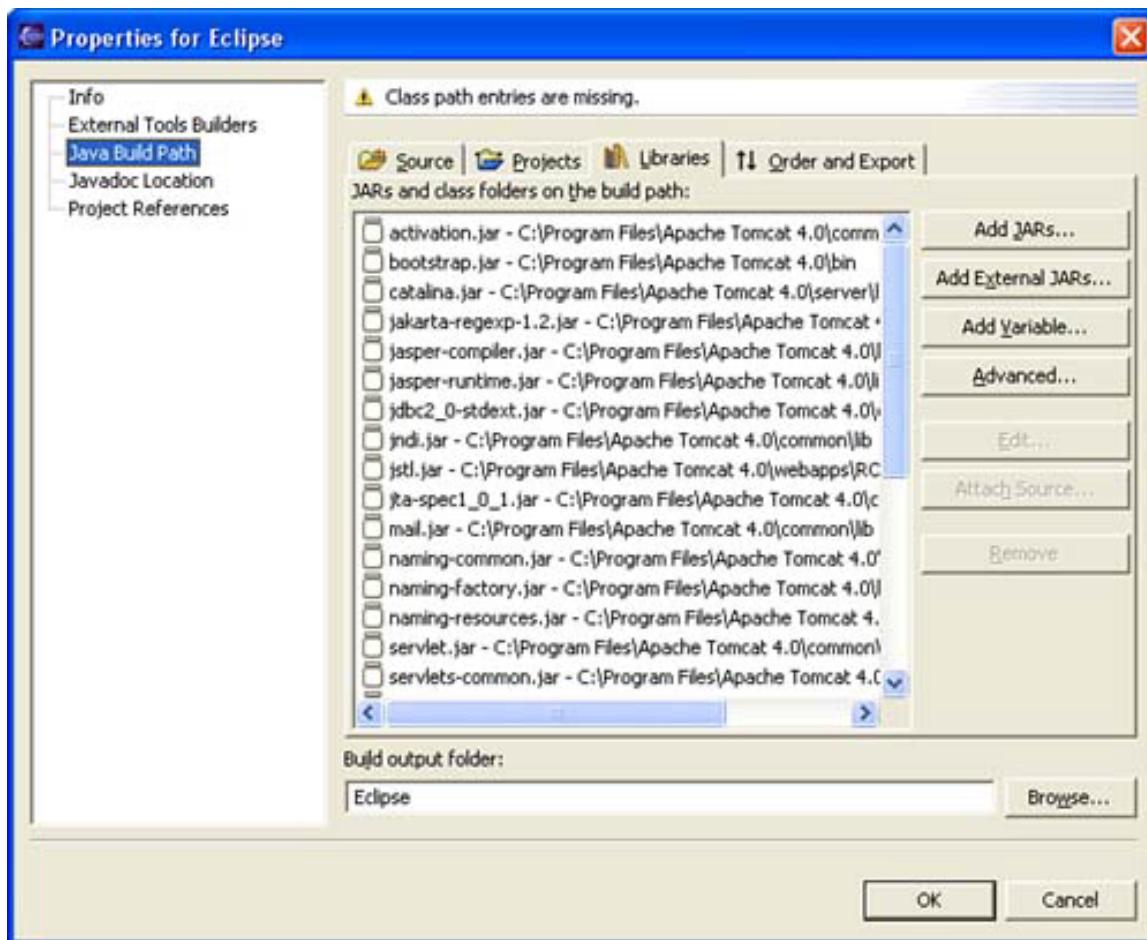
Figure 12.6. The forum project.



Next, make sure that everything is accessible by attempting to rebuild your project. From the Project menu, select Rebuild All. If all goes well, you'll see a progress bar indicating that your project is being recompiled. After the build completes, you should not see any errors. If errors are reported, you must investigate the causes of these errors and correct them. If you are receiving errors, the most likely cause is that the JAR files, which are necessary for Tomcat to run, are not being found by the Eclipse IDE. These JAR files were set up in the Project Properties window and have absolute paths pointing to them. If you installed Tomcat to a folder other than the standard c:\Program Files\Apache Tomcat 4.0, then you must modify the location of each JAR file.

If you installed Tomcat into a nonstandard location, you'll have to adjust the library setting in Project Properties. Begin by right-clicking on the project node on the tree at the left side of the Eclipse IDE. This opens a window that allows you to choose from several tabs. Select the Java Build Path option, and then select the Libraries tab to display a window similar to the one shown in [Figure 12.7](#).

Figure 12.7. Locating the JAR files.



You must now go through each of the entries in the libraries and make sure they point to the correct location of the JAR files. The JAR files are all included within the Tomcat directory. When you've corrected the locations of the JAR files, you should be able to recompile the forum application.

Once you've recompiled the forum application, you should run it to ensure that it actually works. To run the application from Eclipse, click the toolbar item that looks like a running man. This launches the application and allows you to test it.

To test the application, launch a Web browser and attempt to access the forum examples just as you did previously when you were running Tomcat without a debugger. With Tomcat running, you should be able to display the URL

<http://127.0.0.1:8080/example>. If this URL displays, this means you have Tomcat working properly with Eclipse.

If you examine the project file, you may notice that one additional source file was added other than what was already available with the forum application. This source file, named DebugTomcat.Java, starts the Tomcat Web server and allows you to debug it. This short program is shown in Listing 12.4.

Listing 12.4 The DebugTomcat File (DebugTomcat.java)

```
import java.io.*;
import org.apache.catalina.startup.*;

public class DebugTomcat {

    public static void main(String args[]) {
        String a[] = {"start"};
        System.setProperty("catalina.home", "c:\\Program
Files\\Apache Tomcat 4.0");
        try {
            Bootstrap.main(a);
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

As you can see, the main point of this file is to simulate launching Tomcat from the command line. You should also notice the path that is hard-coded into the file. Keep in mind that you will have to change this file path if you stored Tomcat in a nonstandard location.

Now that you have successfully launched Tomcat from the Eclipse IDE, you're ready to begin debugging the forum application.

Creating Your Own Project

If you are going to create your own project, you must add all of the necessary files yourself. You should start with an empty Java project in Eclipse.

Next, import all of the files that you need to compile your tag library. You can import files by right-clicking the project directory and choosing Import. If your .java files are inside packages, create corresponding packages in Eclipse. You create packages by right-clicking the project, selecting New, and choosing Package.

You must also include the starter java file that we discussed in the previous section. This file is named DebugTomcat.java. Modify the path information, as we described earlier. This allows Tomcat to find your JSP and configuration files.

Now you must include all of the JAR files from Tomcat. We usually like to add all the JAR files from Tomcat into the project. They may not all be necessary, depending on which features of Tomcat you are using, but adding them all saves you from running into problems later, when Tomcat can't find them. The Tomcat JAR files are in the `/common` directory. You should include the JSTL JAR files as well, which are usually in the `WEB-INF/lib` directory.

Now that you've set up your project, you are ready to launch Tomcat. If prompted for a class to execute, you should choose the DebugTomcat file we added earlier.

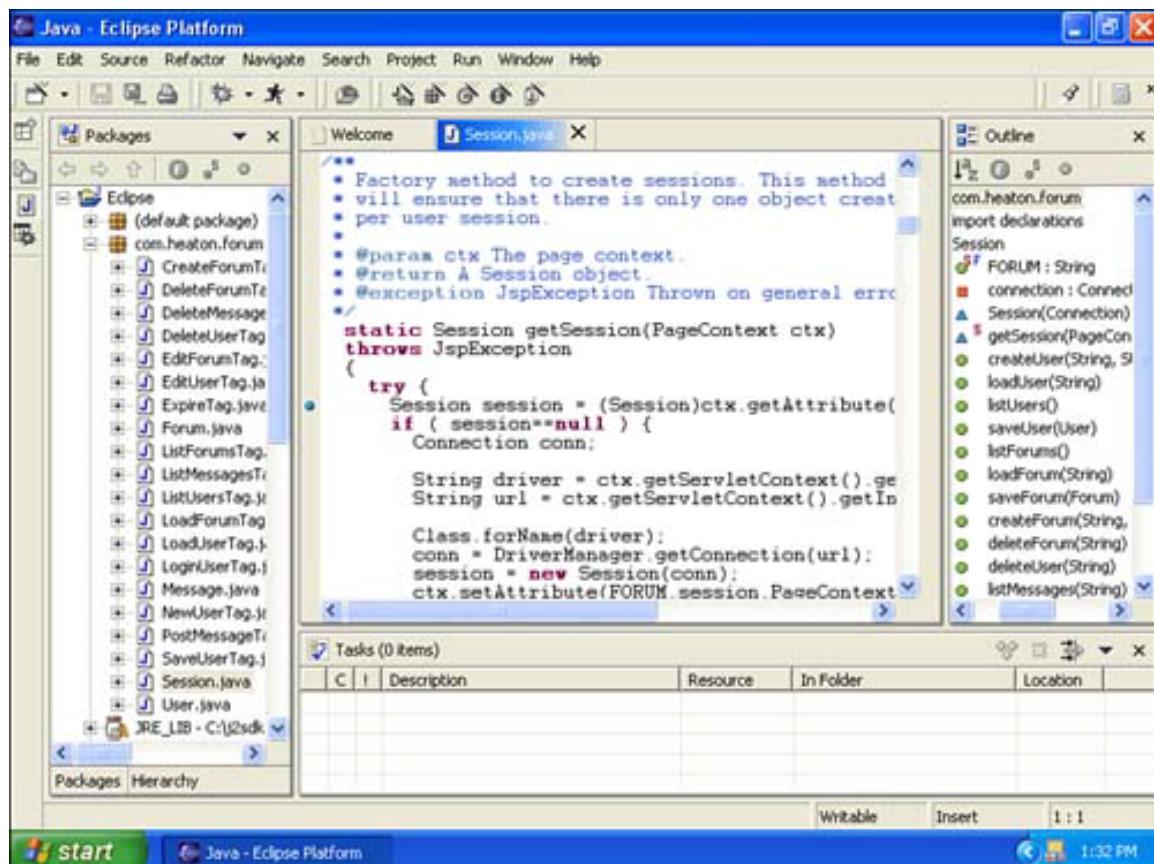
Debugging the Forum Application

You're now ready to begin debugging the forum application. The first step is to place a *breakpoint*. A breakpoint is a specific line in your program that should pause execution when that line is reached. The program will run normally until the breakpoint is hit. At that point, you can pause and examine variables. You can also step through the program one line at a time once you have reached the breakpoint. When you've finished debugging that portion of the program, you can allow the program to continue executing. The program will then continue until it reaches another breakpoint.

For this example, let's place a breakpoint inside one of the source files used for the forum tag library. We'll place the breakpoint in one of the most important source files contained in the forum application: the Session.java file. This file handles database connections used by the forum application. Nearly every tag in the forum tag library uses this source file.

To begin, open the Session.java file. You want to place a breakpoint on the first line of the `getSession()` method. Right-click the gray area to the left of every line in the source file. This will open up a small pop-up menu that allows you to choose to add a breakpoint. Once you have added this breakpoint, your screen should look like the one in [Figure 12.8](#).

Figure 12.8. Setting a breakpoint.



Once you've set the breakpoint, you are ready to execute your program. Keep in mind that you should not click the running-man icon that we used in the previous section; if you use this icon to start your program, breakpoints will not be honored because the program is only running in execute mode. (Execute mode is much faster than debug mode, and you should use it when you do not need breakpoints.) To start the program in debug mode, click the toolbar icon that looks like a bug.

Tomcat launches, and you should see the output from Tomcat near the bottom of the Eclipse screen. This output should look exactly like it does when you start Tomcat in a standalone mode. You can use a browser and open your Web page. You should now open the forum application and proceed to log on normally. Once this process is complete, wait until your program hits the breakpoint.

At this point, you are free to perform any of the debugging operations. This includes examining variables, using watches, or stepping through your program line by line.

WARNING

The Web server may reach your breakpoint and not stop. There could be several reasons for this. The most common is that a compiled JAR file that contains the tag libraries is already in the Tomcat path. If Tomcat has access to the forum.jar file that we previously used, it will use the tags in this JAR file, not the tags in the

debugging session. The result is that Eclipse never gets a chance to see the breakpoints, and thus never stops.

You have now seen how to debug your application. Once an application is debugged, you will likely want to prepare it for distribution. The next section explains how you can package your Web application for easy distribution.

Deploying Web Applications

Installing and packaging your Web application is a critical step. Your application will not be successful if it is not easy to install. Web applications that make use of JSTL and your own custom tag libraries are distributed in exactly the same way as other Web applications. In this section, you'll learn how to create a single archive that stores all of your Web application's files.

Understanding WAR Files

A Web Archive, or WAR file, is a common means of distributing Web applications. A WAR file resembles a Java JAR file: Both have exactly the same format and are based on the Zip file format. A WAR file allows you to package all of the files that are necessary to run your Web application. This includes the JSP pages, the JAR files, and other configuration files. This file is then placed in the Web applications directory (webapps) of a JSP Web server, where it can be executed.

Although a WAR file allows you to store many of the components of your Web application, it cannot store everything. Programs that are completely external to your Web application are not stored in the WAR file. For example, the database in your database connection must be configured separately from the WAR file. The forum application uses a MySQL database to store its data.

While you are not able to store the database as part of the WAR file, the configuration settings used to connect to that database are probably stored in other configuration files. These configuration files will be stored in the WAR file.

Let's look at an example, using the forum application we created in [Chapter 11](#). The forum application is just a directory off the ROOT directory of Tomcat. We'll move this directory to a WAR file. That way, we can distribute the forum application as one single file component that we can place on a Web server.

Creating a WAR file is relatively simple. If you know how to create a JAR file, you already know how to create a WAR file. The procedure for creating a WAR file is identical to that for creating a JAR file. The only difference is that the final output file has a .war extension rather than `.jar`.

If you are using a Unix system, you will likely use the following two jar commands to create the WAR file:

```
jar cvf forum.war /var/jakarta-tomcat-4.0.4/webapps/ROOT/example/
    ch11/forum/*.jsp
jar uvf forum.war /var/jakarta-tomcat-4.0.4/webapps/ROOT/WEB-INF/
```

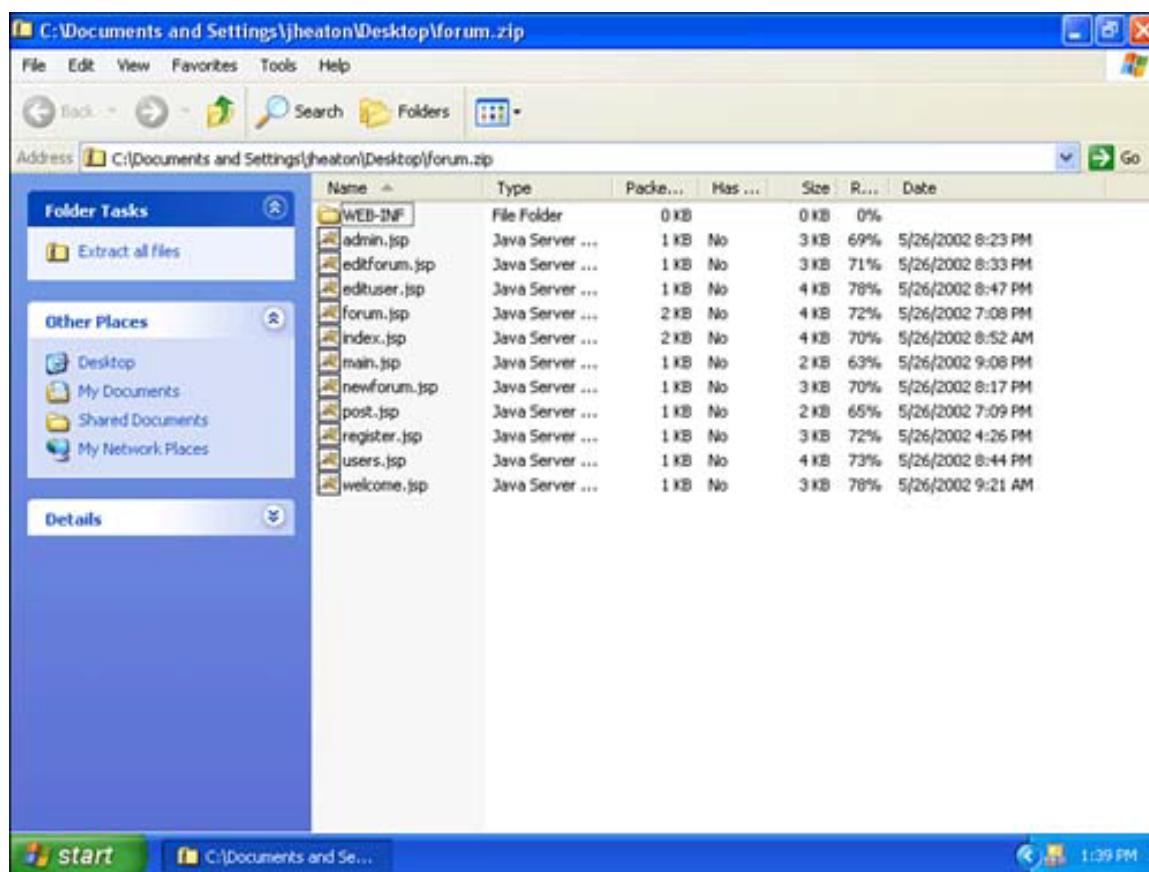
If you are using Microsoft Windows, use the following two jar commands to create the WAR file:

```
jar cvf forum.war C:\Program Files\Apache Tomcat 4.0\webapps\ROOT\
    example\ch11\forum\*.jsp
jar uvf forum.war C:\Program Files\Apache Tomcat 4.0\webapps\ROOT\
    WEB-INF\
```

For both commands, you have to change the directory slightly if you've installed Tomcat in another location.

The WAR file forum.war is shown in [Figure 12.9](#) as a Zip file so that you can see the contents.

Figure 12.9. Creating a Zip file that becomes a WAR file.

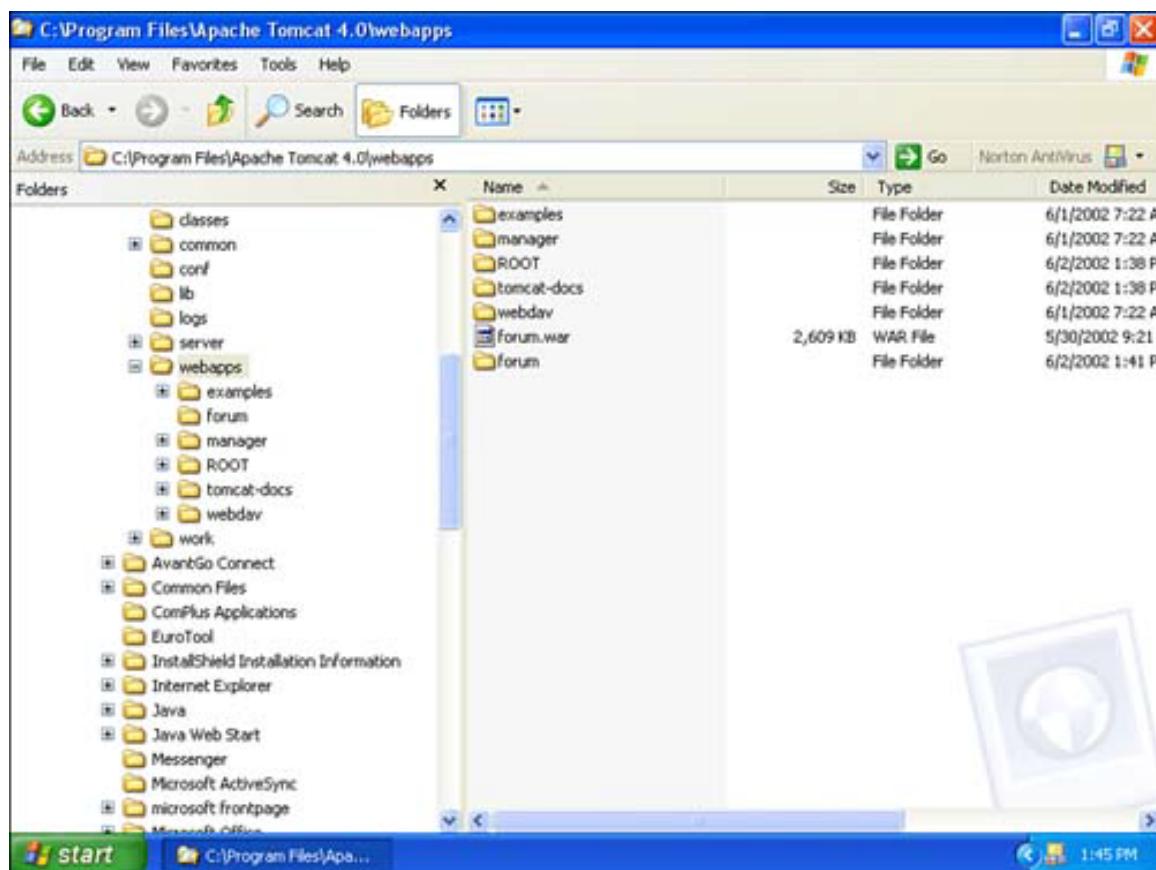


Distributing a WAR File

Now that you've created the Web archive, you must distribute it. Distributing a WAR file is easy. You don't have to make any changes to the Web configuration files to take advantage of the new WAR file. Copy the WAR file to the webapps directory inside Tomcat. When Tomcat is next started, this file will be expanded. The filename of the WAR file becomes part of the URL. For example, to access `forum.war`, you would likely access the URL `http://127.0.0.1:8080/forum`.

WAR files are scanned only when the Web server starts up. If you do not restart the Web server after deploying a WAR file, the Web server will never become aware of the WAR file. [Figure 12.10](#) shows the directory structure of a Web server containing a WAR file that has been expanded by the server.

Figure 12.10. A deployed WAR file.



You have now seen how to properly deploy a WAR file. Now, let's see how to access and test that WAR file. When you place a WAR file in the webapps directory, the contents of that file are automatically made available under a path that corresponds to the name of the archive. For example, you could access the Web archive `forum.jar` from the Web path `http://127.0.0.1:8080/forum`.

Summary

In this chapter, we showed you how to debug and deploy Web applications that make use of JSTL and custom tag libraries. We can package Web applications as WAR files that can be easily integrated with a Web server such as Tomcat. WAR files are a common way to distribute Web applications.

Throughout this book, you learned how to use the JSTL tag library. You are now ready to design and implement Web applications that use the JSTL tags. You have also learned how to develop your own custom tag libraries that are compatible with JSTL. This allows you to create advanced Web applications and encapsulate their unique functionality within custom tag libraries. You can then create advanced Web applications using your own custom tag libraries and using JSTL to tie the entire application together.

Appendix A. JSTL Reference

This appendix contains an alphabetical listing of the JSTL tags. The tags are grouped according to the four tag libraries that make up JSTL. We present each tag, along with the attributes used with that tag.

JSTL supports the notion of twin tag libraries. The tags shown here are from the more commonly used EL library. These tags are capable of handling expressions of the following form:

```
<c:set var="a" value="${100*100}" />
```

In addition to the EL expression language, JSTL includes a second set of tags that are able to process RT expressions. To specify the RT twin library, you must append the suffix `-rt` to the end of the tag library name. For example, the tag `<c:set>` becomes `<c-rt:set>`. The following tag uses the RT expression language:

```
<c-rt:set var="a" value="<%!=100*100%>" />
```

In general, you should use the EL expression language whenever possible. The older RT expression language is included primarily for compatibility with older code.

The Core Tags

The core tags perform operations that form the foundation of the JSTL tag library. Using the core tags, you can set scoped variables, iterate over collections, and handle exceptions.

The `<c:catch>` Tag

The `<c:catch>` tag is used to allow JSP pages to handle exceptions that might be raised by any of the code contained in the body of the `<c:catch>` tag.

```
<c:catch var="e">
    ... Program code that may throw an exception ...
</c:catch>
```

The `<c:catch>` tag has one attribute:

Attribute	Required	Purpose
<code>var</code>	Y	Specifies the scoped variable that should receive the exception.

The <c:choose> Tag

The `<c:choose>` tag is used to form the body of a mutual exclusion. The tags `<c:when>` and `<c:otherwise>` should be embedded inside the `<c:choose>` tag. When the `<c:choose>` block is reached, only one of the `<c:when>` tag blocks will be executed. If none of the `<c:when>` tags evaluates to true, then the `<c:otherwise>` tag, if present, will be executed. The `<c:choose>` block works much like the switch, case, and default statements in the Java programming language. The `<c:choose>` tag has no attributes.

```
<c:choose>
    body content (<when> and <otherwise> subtags)
</c:choose>
```

The <c:forEach> Tag

The `<c:forEach>` tag is used to loop in several different ways. The most common use for this tag is to loop over a collection. It is also possible to make the `<c:forEach>` tag operate similar to the Java for statement. By including begin, end, and step attributes, you can specify that the `<c:forEach>` tag loop a certain number of times.

```
// Syntax 1: Iterate over a collection of objects
<c:forEach [var="varName"] items="collection"
[varStatus="varStatusName"]
[begin="begin"] [end="end"] [step="step"]>
body content
</c:forEach>
// Syntax 2: Iterate a fixed number of times
<c:forEach [var="varName"]
[varStatus="varStatusNamev"]
begin="begin" end="end" [step="step"]>
body content
</c:forEach>
```

The `<c:forEach>` tag has the following attributes:

Attribute	Required	Purpose
<code>begin</code>	N	If items is specified, iteration begins at this item. If items is not specified, iteration begins with the index at this value.
<code>end</code>	N	If items is specified, iteration ends at this item. If items is not specified, iteration ends with the index at this value.
<code>items</code>	N	A collection of items to iterate.
<code>step</code>	N	The amount added to the index through each iteration.
<code>var</code>	N	Holds the current item in the iteration.
<code>varStatus</code>	N	Allows you to specify a <code>javax.servlet.jsp.jstl.core.LoopTagStatus</code> object to give you information about the iteration as it processes.

The <c:forTokens> Tag

The `<c:forTokens>` tag works much like the `<c:forEach>` tag, except that the `<c:forTokens>` tag iterates over a delineated string rather than a collection. You are allowed to specify a delineator for the string.

```
<c:forTokens items="stringOfTokens" delims="delimiters"
[var="varName"]
[varStatus="varStatusName"]
[begin="begin"] [end="end"] [step="step"]>
body content
</c:forEach>
```

The `<c:forTokens>` tag has these attributes:

Attribute	Required	Purpose
<code>begin</code>	N	Iteration will begin at this token. The first token has an index of 0.
<code>delims</code>	N	The delimiters used to parse with.
<code>end</code>	N	Iteration will end at this token.
<code>items</code>	N	A string of tokens to iterate through.
<code>step</code>	N	The amount added to the index through each iteration.
<code>var</code>	N	Holds the current item in the iteration.
<code>varStatus</code>	N	Allows you to specify a <code>javax.servlet.jsp.jstl.core.LoopTagStatus</code> object to give you information about the iteration as it processes.

The <c:if> Tag

The `<c:if>` tag allows you to conditionally execute a block of tags if the condition specified in the `<c:if>` tag is met. It is important to note that JSTL does not include an else tag. If you need to perform an if statement with an else, you must use a `<c:choose>` tag with a `<c:otherwise>` tag.

```
// <!-- Syntax 1: Without body content -->
<c:if test="testCondition"
      var="varName" [scope="{page|request|session|application}"]/>
// <!-- Syntax 2: With body content -->
<c:if test="testCondition"
      [var="varName"] [scope="{page|request|session|application}"]>
body content
</c:if>
```

The `<c:if>` tag has the following attributes:

Attribute	Required	Purpose
<code>test</code>	Y	Specifies the test that is to be performed.

<code>scope</code>	N	Specifies the scope; the default is page.
<code>var</code>	N	Specifies the scoped variable that should receive the result of the compare.

The `<c:import>` Tag

The `<c:import>` tag is used to read data from an external URL address. When the `<c:import>` tag is executed, the contents of the specified URL are downloaded and stored in a string specified in the `<c:import>` tag. This tag can be useful for retrieving XML data from Web addresses.

```
// Syntax 1: Resource content inlined or exported as a String object
<c:import url="url" [context="context"]
[var="varName"] [scope="{page|request|session|application}"]
[charEncoding="charEncoding"]>
optional body content for <c:param> subtags
</c:import>
// Syntax 2: Resource content exported as a Reader object
<c:import url="url" [context="context"]
varReader="varReaderName"
[charEncoding="charEncoding"]>
body content where varReader is consumed by another action
</c:import>
```

The `<c:import>` tag has the following attributes:

Attribute	Required	Purpose
<code>charEncoding</code>	N	Allows you to specify the character encoding (for example, UTF16).
<code>context</code>	N	Specifies the base URL that will be used to resolve a relative URL given by the <code>url</code> attribute.
<code>scope</code>	N	Specifies the scope for the scoped variable referenced by the <code>var</code> attribute.
<code>url</code>	Y	Specifies the URL from which the import is to take place.
<code>var</code>	N	Specifies a variable that will receive the output from the specified URL.
<code>varReader</code>	N	A Reader object to output to.

The `<c:otherwise>` Tag

The `<c:otherwise>` tag works much like the `default` statement in Java. The `<c:otherwise>` tag can be used only inside a `<c:choose>` tag's body. Use the `<c:otherwise>` tag to specify a block of code that is to be executed if none of the `<c:when>` tags (which are also contained inside the body of the `<c:choose>` tag) are executed. The `<c:otherwise>` tag has no attributes.

```
<c:otherwise>
    conditional block
</c:otherwise>
```

The <c:out> Tag

The `<c:out>` tag is used to output data to the current page. Using this tag, you can display the results of variables and other expressions.

```
// Syntax 1: Without a body
<c:out value="value" [escapeXml="{true|false}"]
[default="defaultValue"] />
// Syntax 2: With a body
<c:out value="value" [escapeXml="{true|false}"]>
default value
</c:out>
```

The `<c:out>` tag has the following attributes:

Attribute	Required	Purpose
<code>default</code>	N	Specifies the value that will be displayed if the specified value evaluates to null.
<code>escapeXml</code>	N	Uses a value of true to escape any special characters, or false to display text as is. The default is true.
<code>value</code>	Y	The value that is to be displayed.

The <c:param> Tag

The `<c:param>` tag is used to specify a parameter to the `<c:redirect>`, `<c:url>`, and `<c:import>` tags. These parameters will be appended to the URL as URL-based parameters.

```
// Syntax 1: Parameter value specified in attribute value
<c:param name="name" value="value"/>
// Syntax 2: Parameter value specified in the body content
<c:param name="name">
parameter value
</c:param>
```

The `<c:param>` tag has the following attributes:

Attribute	Required	Purpose
<code>name</code>	Y	Specifies the name of this parameter.
<code>value</code>	N	Specifies the value of this parameter.

The <c:redirect> Tag

The <c:redirect> tag is used to completely move the Web user to another page. This allows you to redirect the user before the current page is displayed. This can be useful for many purposes, such as redirecting users away from a page to which they do not have access. If the <c:redirect> tag is used, no other processing will occur on the page.

```
// Syntax 1: Without body content
<c:redirect url="value"/>
// Syntax 2: With body content to specify query string parameters
<c:redirect url="value"/>
<c:param> subtags
</c:redirect>
```

The <c:redirect> tag has the following attributes:

Attribute	Required	Purpose
context	N	Specifies the base URL that will be used to resolve a relative URL given by the url attribute.
url	Y	Specifies the URL that the user is taken to.

The <c:remove> Tag

The <c:remove> tag is used to delete a scoped variable. Scoped variables are created by <c:set> and other tags. The <c:remove> tag allows you to reclaim the space occupied by the specified scoped variable.

```
<c:remove [scope="{page|request|session|application}"] var="var"/>
```

The <c:remove> tag has the following attributes:

Attribute	Required	Purpose
scope	N	Specifies the scope of the var variable; defaults to page.
var	Y	Specifies the scoped variable that is to be removed.

The <c:set> Tag

The <c:set> tag is used to assign the specified expression to the specified scoped variable. This allows the <c:set> tag to be used to specify the values of scoped variables. Once a scoped variable is no longer needed, you can remove it by calling the <c:remove> tag.

```
// Syntax 1: Set using value attribute
<c:set value="value"
var="varName" [scope="{page|request|session|application}"]/>
// Syntax 2: Set the value of a scoped variable using body content
```

```

<c:set var="varName" [scope="{page|request|session|application}"]>
body content
</c:set>
// Syntax 3: Set the value of a target object using value attribute
<c:set value="value"
target="target" property="propertyName"/>
// Syntax 4: Set the value of a target object using body text
<c:set target="target" property="propertyName">
body content
</c:set>

```

The `<c:set>` tag has the following attributes:

Attribute	Required	Purpose
<code>property</code>	N	If target specifies an object, then property specifies the property.
<code>scope</code>	N	Specifies the scope of the var variable; defaults to page.
<code>target</code>	N	An object that will have one of its properties set.
<code>value</code>	Y	Specifies the value that is to be set to the scoped variable var.
<code>var</code>	Y	Specifies the scoped variable that is to be set.

The `<c:url>` Tag

The `<c:url>` tag is used to construct a URL from a specified context URL and optional parameters (specified using the `<c:param>` tag).

```

// Syntax 1: Without body content
<c:url url="value"
[var="varName"] [scope="{page|request|session|application}"]/>
// Syntax 2: With body content to specify query string parameters
<c:url url="value"
[var="varName"] [scope="{page|request|session|application}"]>
<c:param> subtags
</c:url>

```

The `<c:url>` tag has the following parameters:

Attribute	Required	Purpose
<code>context</code>	N	Specifies the base URL that will be used to resolve a relative URL given by the value attribute.
<code>scope</code>	N	Specifies the scope for the scoped variable referenced by the var attribute. The default is page.
<code>value</code>	Y	Specifies the URL from which the import is to take place.
<code>var</code>	N	Specifies a variable that will receive the output from the specified URL.

The <c:when> Tag

The <c:when> tag is used as part of a <c:choose> block. The <c:choose> tag will be examined until one <c:when> tag is found that contains a test that is satisfied.

```
<c:when test="testCondition">  
body content  
</c:when>
```

The <c:when> tag has one attribute:

Attribute	Required	Purpose
test	Y	Specifies the test that is to be performed.

The I18N Tags

Java contains extensive support for internationalization, or I18N. Multilingual programming is supported by allowing strings to be stored in language-specific Java resource bundles. These bundles can be exchanged as language needs vary. JSTL also includes formatting tags that allow you to properly format numbers, dates, and currencies for specific locales.

The <fmt:bundle> Tag

The <fmt:bundle> tag is used to load the specified bundle and use it as the default bundle for any <fmt:message> tags that fall inside the <fmt:bundle> block. If you want to load a bundle to a scoped variable, you should use the <fmt:setBundle> tag.

```
<fmt:bundle basename="basename" prefix="prefix"/>  
</fmt:setBundle>
```

The <fmt:bundle> tag has the following attributes:

Attribute	Required	Purpose
basename	Y	Specifies the base name of the resource bundle that is to be loaded.
prefix	N	Specifies a prefix that will be added to every key attribute specified in the <fmt:message> tags.

The <fmt:formatDate> Tag

The <fmt:formatDate> tag is used to format dates in a variety of standard forms.

```
<fmt:formatDate [value="date"]
```

```
[type="{time|date|both}"]
[dateStyle="{default|short|medium|long|full}"]
[timeStyle="{default|short|medium|long|full}"]
[pattern="customPattern"]
[timeZone="timeZone"]
[var="varName"]
[scope="{page|request|session|application}"]/>
```

The `<fmt:formatDate>` tag uses the following attributes:

Attribute	Required	Purpose
<code>dateStyle</code>	N	A predefined formatting style for dates. Applied only when formatting a date, or both a date and time (that is, if type is missing or is equal to date or both); ignored otherwise.
<code>pattern</code>	N	A custom formatting style for dates and times.
<code>scope</code>	N	The scope of var; the default is page.
<code>timeStyle</code>	N	A predefined formatting style for times. Applied only when formatting a time, or both a date and time (that is, if type is equal to time or both); ignored otherwise.
<code>timeZone</code>	N	Specifies the time zone in which to represent the formatted time.
<code>type</code>	N	Specifies whether only the time, the date, or both the time and date components of the given date are to be formatted. The default is date.
<code>value</code>	Y	The date and/or time that should be formatted.
<code>var</code>	N	The name of the exported scoped attribute that stores the formatted result as a String.

The `<fmt:formatNumber>` Tag

The `<fmt:formatNumber>` tag is used to format dates in a variety of standard forms.

```
// Syntax 1: Without a body
<fmt:formatNumber value="numericValue"
  [type="{number|currency|percent}"]
  [pattern="customPattern"]
  [currencyCode="currencyCode"]
  [currencySymbol="currencySymbol"]
  [groupingUsed="{true|false}"]
  [maxIntegerDigits="maxIntegerDigits"]
  [minIntegerDigits="minIntegerDigits"]
  [maxFractionDigits="maxFractionDigits"]
  [minFractionDigits="minFractionDigits"]
  [var="varName"]
  [scope="{page|request|session|application}"]/>
// Syntax 2: With a body to specify the numeric value to be
formatted
<fmt:formatNumber [type="{number|currency|percent}"]
  [pattern="customPattern"]
  [currencyCode="currencyCode"]
```

```
[currencySymbol="currencySymbol"]
[groupingUsed="{true|false}"]
[maxIntegerDigits="maxIntegerDigits"]
[minIntegerDigits="minIntegerDigits"]
[maxFractionDigits="maxFractionDigits"]
[minFractionDigits="minFractionDigits"]
[var="varName"]
[scope="{page|request|session|application}"]>
    numeric value to be formatted
</fmt:formatNumber>
```

The `<fmt:formatNumber>` tag uses the following attributes:

Attribute	Required	Purpose
<code>currencyCode</code>	N	String ISO 4217 currency code. Applied only when formatting currencies (that is, if type is equal to currency); ignored otherwise.
<code>currencySymbol</code>	N	String Currency symbol. Applied only when formatting currencies (that is, if type is equal to currency); ignored otherwise.
<code>groupingUsed</code>	N	Specifies whether the formatted output will contain any grouping separators. The default is true.
<code>maxFractionDigits</code>	N	The maximum number of digits in the fractional portion of the formatted output.
<code>maxIntegerDigits</code>	N	The maximum number of digits in the integer portion of the formatted output.
<code>minFractionDigits</code>	N	The minimum number of digits in the fractional portion of the formatted output.
<code>minIntegerDigits</code>	N	The minimum number of digits in the integer portion of the formatted output.
<code>pattern</code>	N	String Custom formatting pattern. Applied only when formatting numbers (that is, if type is missing or is equal to number); ignored otherwise.
<code>scope</code>	N	The scope of var. The default is page.
<code>type</code>	N	Specifies whether the value is to be formatted as number, currency, or percentage. The default is number.
<code>value</code>	N	The String or Number Numeric value to be formatted.
<code>var</code>	N	The name of the exported scoped attribute that stores the formatted result as a String.

The `<fmt:message>` Tag

The `<fmt:message>` tag is used to insert multilingual text into JSP pages. This tag references strings stored in resource bundles. If the `<fmt:message>` tag is inside a `<fmt:bundle>` block, then `<fmt:message>` will use `<fmt:bundle>` to determine which

resource bundle to use. If the `<fmt:message>` tag is outside a `<fmt:bundle>` block, then `resourceBundle` must be scoped variable obtained from a previous call to `<fmt:setBundle>`. It can also use a configured localization context.

```
// Syntax 1: Without body content
<fmt:message key="messageKey"
[bundle="resourceBundle"]
[var="varName"]
[scope="{page|request|session|application}"]/>
// Syntax 2: With a body to specify message parameters
<fmt:message key="messageKey"
[bundle="resourceBundle"]
[var="varName"]
[scope="{page|request|session|application}"]>
<fmt:param> subtags
</fmt:message>
// Syntax 3: With a body to specify key and optional message
parameters
<fmt:message [bundle="resourceBundle"]
[var="varName"]
[scope="{page|request|session|application}"]>
key
optional <fmt:param> subtags
</fmt:message>
```

The `<fmt:message>` tag has the following attributes:

Attribute	Required	Purpose
<code>bundle</code>	N	Specifies the resource bundle that is to be used. If this tag occurs from within a <code><fmt:bundle></code> tag, then no bundle needs to be specified.
<code>key</code>	N	Specifies the lookup key that identifies the desired string inside the bundle.
<code>scope</code>	N	Specifies the scope of the scoped variable referenced by the <code>var</code> attribute. If the <code>var</code> attribute is not needed, then the <code>scope</code> attribute should not be specified. The default is <code>page</code> .
<code>var</code>	N	Specifies a scoped variable that will receive the value of the string that is being looked up. If no <code>var</code> attribute is specified, then the output will be written to the page.

The `<fmt:param>` Tag

The `<fmt:param>` tag is used to specify parameters that will fill in the `<fmt:message>` string.

```
// Syntax 1: Value specified via attribute value
<fmt:param value="messageParameter"/>
// Syntax 2: Value specified via body content
<fmt:param>
```

```

body content
</fmt:param>

```

The `<fmt:param>` tag has one attribute. Only 10 parameters may be specified.

Attribute	Required	Purpose
<code>value</code>	N	Specifies the value that is to be inserted for the parameter.

The `<fmt:parseDate>` Tag

The `<fmt:parseDate>` tag is used to transform a string into a Java date.

```

// Syntax 1: Without a body
<fmt:parseDate value="dateString"
    [type="{time|date|both}"]
    [dateStyle="{default|short|medium|long|full}"]
    [timeStyle="{default|short|medium|long|full}"]
    [pattern="customPattern"]
    [timeZone="timeZone"]
    [parseLocale="parseLocale"]
    [var="varName"]
    [scope="{page|request|session|application}"]/>
// Syntax 2: With a body to specify the date value to be parsed
<fmt:parseDate [type="{time|date|both}"]
    [dateStyle="{default|short|medium|long|full}"]
    [timeStyle="{default|short|medium|long|full}"]
    [pattern="customPattern"]
    [timeZone="timeZone"]
    [parseLocale="parseLocale"]
    [var="varName"]
    [scope="{page|request|session|application}"]>
    date value to be parsed
</fmt:parseDate>

```

The `<fmt:parseDate>` tag has the following attributes:

Attribute	Required	Purpose
<code>dateStyle</code>	N	The style for the date. Should be default, short, long, medium, or full. The default is default. This attribute uses DateFormat semantics.
<code>parseLocale</code>	N	Specifies the locale that should be used. This attribute should be of type <code>String</code> or <code>java.util.Locale</code> .
<code>pattern</code>	N	Specifies how the date is to be parsed using a pattern.
<code>timeStyle</code>	N	The style for the time. Should be default, short, long, medium, or full. The default is default.
<code>timeZone</code>	N	The zone that the parsed time is from. This value should be of type <code>String</code> or <code>java.util.TimeZone</code> .
<code>type</code>	N	The type of date to parse. Should be time, date, or both. The

		default is date.
value	N	The string to be parsed.
var	N	The name of the exported scoped attribute that stores the parsed result (of type <code>java.lang.Number</code>).

The <fmt:parseNumber> Tag

The `<fmt:parseNumber>` tag is used to parse strings into numeric values. If the value specified cannot be transformed into a numeric value, then an exception will be thrown.

```
// Syntax 1: Without a body
<fmt:parseNumber value="numericValue"
  [type="{number|currency|percent}"]
  [pattern="customPattern"]
  [parseLocale="parseLocale"]
  [integerOnly="{true|false}"]
  [var="varName"]
  [scope="{page|request|session|application}"]/>
// Syntax 2: With a body to specify the numeric value to be parsed
<fmt:parseNumber [type="{number|currency|percent}"]
  [pattern="customPattern"]
  [parseLocale="parseLocale"]
  [integerOnly="{true|false}"]
  [var="varName"]
  [scope="{page|request|session|application}"]>
  numeric value to be parsed
</fmt:parseNumber>
```

The `<fmt:parseNumber>` tag has the following attributes:

Attribute	Required	Purpose
integerOnly	N	Specifies whether just the integer portion of the given value should be parsed. Defaults to false.
parseLocale	N	The locale whose default formatting pattern (for numbers, currencies, or percentages, respectively) is to be used during the parse operation, or to which the pattern specified via the pattern attribute (if present) is applied.
pattern	N	Custom formatting pattern that determines how the string in the value attribute is to be parsed. Applied only when parsing numbers (that is, if type is missing or is equal to number); ignored otherwise.
scope	N	The scope of var. Defaults to page.
type	N	Specifies whether the string in the value attribute should be parsed as number, currency, or percentage. Defaults to number.
value	N	The string to be parsed.
var	N	The name of the exported scoped attribute that stores the parsed result (of type <code>java.lang.Number</code>).

The <fmt:requestEncoding> Tag

The <fmt:requestEncoding> tag is used to specify the encoding method used in the form that posted to the current page. This allows JSTL to properly determine the parameters that were posted to the current page.

```
<fmt:requestEncoding [value="charsetName"] />
```

The <fmt:requestEncoding> tag has one attribute:

Attribute	Required	Purpose
value	N	The name of the character encoding to be applied when decoding request parameters.

The <fmt:setBundle> Tag

The <fmt:setBundle> tag is used to load a Java resource bundle into a JSTL scoped variable. The scoped variable can then be provided to future calls to the <fmt:message> tag.

```
<fmt:setBundle basename="basename"
 [var="varName"]
 [scope="{page|request|session|application}"]>
```

The <fmt:setBundle> tag contains the following attributes:

Attribute	Required	Purpose
basename	Y	Specifies the base name of the resource bundle that is to be loaded in the form or fully qualified classname.
scope	N	Specifies the scope of the variable that is contained in the var attribute. Defaults to page.
var	N	Specifies a scoped variable that will hold the newly instantiated resource bundle.

The <fmt:setLocale> Tag

The <fmt:setLocale> tag is used to specify the current locale for the Web application.

```
<fmt:setLocale value="locale"
 [variant="variant"]
 [scope="{page|request|session|application}"]/>
```

The <fmt:setLocale> tag has the following attributes:

Attribute	Required	Purpose
-----------	----------	---------

value	Y	The locale species a two-part code that represents the ISO-639 language code and an ISO-3166 country code (for example, en-US).
scope	N	Specifies the scope of this locale setting. The default is page.
variant	N	Specifies a vendor- or browser-specific variant of the language referenced by value. For more information, refer to the java.util.Locale JavaDocs.

The <fmt:setTimeZone> Tag

The <fmt:setTimeZone> tag is used to specify the time zone for the Web application.

```
<fmt:setTimeZone value="timeZone"
[var="varName"]
[scope="{page|request|session|application}"]
```

The <fmt:setTimeZone> tag has the following attributes:

Attribute	Required	Purpose
scope	N	Specifies the scope of scoped variable referenced by var. The default is page.
value	Y	Specifies the time zone that should be used by the tags enclosed in the body of this tag. See the java.util.TimeZone JavaDoc for formats.
var	N	Specifies the scoped variable that is to receive the time zone.

The <fmt:timeZone> Tag

The <fmt:timeZone> tag is used to specify a time zone that will be used by all tags inside the body of the <fmt:timeZone> tag. The <fmt:timeZone> tag has one attribute:

Attribute	Required	Purpose
value	Y	Specifies the time zone that should be used by the tags enclosed in the body of this tag. See the java.util.TimeZone JavaDoc for formats.

The Relational Database Tags (SQL)

The relational database tags allow a Web application to access a SQL database. You must have a JDBC driver and know the correct URL to use to access this database. You can use the SQL tags to perform queries, updates, and use transactions.

The SQL tags are intended primarily for rapid prototyping or very small Web applications. For most nontrivial Web applications, we suggest that you isolate your database access into tag libraries of your own. [Chapter 11](#) shows how to do this.

The <sql:dateParam> Tag

If you are going to specify a parameter to the <sql:query> or <sql:update> tags and that parameter is a date, you must use the <sql:dateParam> tag. If the parameter is not a date, then use the <sql:param> tag.

```
<sql:dateParam value="value" type="[timestamp|time|date]" />
```

The <sql:dateParam> tag has the following attributes:

Attribute	Required	Purpose
value	Y	The value to be inserted for this parameter.
type	N	Must be timestamp, time, or date. The default is timestamp.

The <sql:param> Tag

If you are going to specify a parameter to the <sql:query> or <sql:update> tags and that parameter is any value but a date, you must use the <sql:param> tag. If the value is a date, then you should use the <sql:dateParam> tag.

```
// Syntax 1: Parameter value specified in attribute value
<sql:param value="value"/>
// Syntax 2: Parameter value specified in the body content
<sql:param>
parameter value
</sql:param>
```

The <sql:param> tag has one attribute:

Attribute	Required	Purpose
value	N	The value to be inserted for this parameter.

The <sql:query> Tag

A query, as defined by JSTL, is any SQL command that is going to return records. The most common SQL commands that return records are the SELECT SQL command and calls to stored procedures. Using the <sql:query> tag, you can submit a query and receive back a collection of records. You can then process these records using the core iteration tags, such as <c:forEach>.

```
// Syntax 1: Without body content
<sql:query sql="sqlQuery"
```

```

var="varName" [scope="{page|request|session|application}"]
[dataSource="dataSource"]
[maxRows="maxRows"]
[startRow="startRow"]/>
// Syntax 2: With a body to specify query arguments
<sql:query sql="sqlQuery"
var="varName" [scope="{page|request|session|application}"]
[dataSource="DataSource"]
[maxRows="maxRows"]
[startRow="startRow"]>
<sql:param> actions
</sql:query>
// Syntax 3: With a body to specify query and optional query parameters
<sql:query var="varName"
[scope="{page|request|session|application}"]
[dataSource="DataSource"]
[maxRows="maxRows"]
[startRow="startRow"]>
query
optional <sql:param> actions
</sql:query>
```

The `<sql:query>` tag has the following attributes:

Attribute	Required	Purpose
<code>dataSource</code>	N	Specifies the data source to be used with this query.
<code>maxRows</code>	N	Specifies the maximum number of rows to be returned.
<code>scope</code>	N	Specifies the scope for the scoped variable referenced by the <code>var</code> attribute. Defaults to page.
<code>sql</code>	N	Specifies the SQL command that is to be executed.
<code>startRow</code>	N	Specifies the starting row that should be returned in the result set.
<code>var</code>	N	Specifies the scoped variable that will receive the results of this query.

The `<sql:setDataSource>` Tag

To use a `<sql:update>`, `<sql:query>`, or `<sql:transaction>` tag, you must have a data source. The `<sql:setDataSource>` tag is used to open a connection to a data source. You must know both the driver name and access URL for the data source. Consult the documentation associated with your database to see what these values are.

```

<sql:setDataSource var="varName"
[scope="{page|request|session|application}"]
[driver="driverClassName"]
[url="jdbcUrl"]
[user="userName"]/>
```

The `<sql:setDataSource>` tag has the following attributes:

Attribute	Required	Purpose
dataSource	N	The data source that should be used.
driver	N	The classname of the JDBC data source that should be used.
scope	N	The scope of the scoped variable referenced by the attribute var. The default is page.
url	N	The URL of the data source.
password	N	The password used to establish the connection.
user	N	The user that should be used to log into the data source.
var	Y	The scoped variable that will hold the newly created data source.

The <sql:transaction> Tag

Often, when updating a database, you must perform several updates to complete one transaction. Usually, it is preferable that if a single operation fails, the entire transaction will fail. This is the purpose of the <sql:transaction> tag. Any <sql:update> tags inside the body of the <sql:transaction> tag will operate as one transaction. If any one of these <sql:update> tags fails, the entire transaction fails.

```
[isolation=isolationLevel]
<sql:query> and <sql:update> statements
</sql:transaction>
```

The <sql:transaction> tag has the following attributes:

Attribute	Required	Purpose
dataSource	N	Specifies the data source to be used with this transaction.
isolationLevel	N	Specifies the isolation level for this transaction— TRANSACTION_READ_COMMITTED, TRANSACTION_READ_UNCOMMITTED, TRANSACTION_REPEATABLE_READ, or TRANSACTION_SERIALIZABLE).

The <sql:update> Tag

JSTL defines an update as a SQL command that does not return a result set. SQL updates usually modify the database in some way, though this is not an absolute requirement. The changes made by the <sql:update> tag are immediate, unless the tag is being used as part of a <sql:transaction> block.

```
// Syntax 1: Without body content
<sql:update sql="sqlUpdate"
[dataSource="DataSource"]
[var="varName"] [scope="{page|request|session|application}"]/>
// Syntax 2: With a body to specify update parameters
<sql:update sql="sqlUpdate"
```

```

[dataSource="DataSource"]
[var="varName"] [scope="{page|request|session|application}"]>
<sql:param> actions
</sql:update>
// Syntax 3: With a body to specify update statement and optional
update parameters
<sql:update [dataSource="DataSource"]>
[var="varName"] [scope="{page|request|session|application}"]>
update statement
optional <sql:param> actions
</sql:update>

```

The `<sql:update>` tag has the following attributes:

Attribute	Required	Purpose
<code>dataSource</code>	N	Specifies the data source to be used with this command.
<code>scope</code>	N	Specifies the scope for the scoped variable referenced by the <code>var</code> attribute. Defaults to page.
<code>sql</code>	N	Specifies the SQL command that should be executed.
<code>var</code>	N	Specifies the scoped variable that will receive the results of this command.

The XML Tags

XML is becoming a common form of data encountered on the Internet. JSTL makes available several XML tags that allow XML documents to be formatted and processed. Most of the XML tags use XPath expressions to specify the operation you want to perform. XPath is covered in [Chapter 8](#).

The `<x:choose>` Tag

The `<x:choose>` tag is used to specify a mutual exclusion block. Inside the `<x:choose>` tag's body, `<x:when>` and `<x:otherwise>` tags reside. The first `<x:choose>` tag that is satisfied will execute the tags inside the `<x:choose>` tag's body. If no `<x:choose>` tags are satisfied, then the `<x:otherwise>` tag, if present, will be executed. The `<x:choose>` tag has no attributes.

```

<x:choose>
body content (<x:when> and <x:otherwise> subtags)
</x:choose>

```

The `<x:forEach>` Tag

The `<x:forEach>` tag allows you to iterate through child nodes of an XML document. The `<x:forEach>` tag works much like the `<c:forEach>` tag, in that the body is executed

for each item in the collection. As the loop progresses, each item is copied to the var attribute.

```
<x:forEach [var="varName"] select="XpathExpression">  
    body content  
</x:forEach>
```

The `<x:forEach>` tag has the following attributes:

Attribute	Required	Purpose
<code>select</code>	Y	An XPath expression to iterate over.
<code>var</code>	N	Specifies the scoped variable that will hold each iteration.

The `<x:if>` select Tag

The `<x:if>` tag is used to perform basic logic on the XML document that has been parsed. Using the `<x:if>` statement, you can examine individual elements in the node you specify using the XPath data in the select attribute.

```
// Syntax 1: Without body, result copied to var  
    <x:if select="XpathExpression"  
        var="varName" [scope="{page|request|session|application}"]/>  
// Syntax 2: With conditional body content  
    <x:if select="XpathExpression"  
        [var="varName"] [scope="{page|request|session|application}"]>  
        body content  
</x:if>
```

The `<x:if>` tag has the following attributes:

Attribute	Required	Purpose
<code>scope</code>	N	Specifies the scope for the scoped variable referenced by the var attribute. The default is page.
<code>select</code>	Y	An XPath expression that should be evaluated (true or false).
<code>var</code>	N	Specifies the scoped variable that can be set to the result of the if statement.

The `<x:otherwise>` Tag

The `<x:otherwise>` tag is used as part of an `<x:choose>` block. If none of the `<x:when>` tags is satisfied, then the `<x:choose>` block will execute the `<x:otherwise>` tag. The `<x:otherwise>` tag does not have any attributes.

```
<x:otherwise>  
    conditional block
```

```
</x:otherwise>
```

The <x:out> Tag

The `<x:out>` tag is used to display individual elements from the XML document.

```
<x:out select="XpathExpression" [escapeXml="{true|false}"] />
```

The `<x:out>` tag has the following attributes:

Attribute	Required	Purpose
<code>escapeXml</code>	N	Specifies whether the displayed data should have its special characters escaped. Defaults to true.
<code>select</code>	Y	An XPath expression that you want to display.

The <x:param> Tag

The `<x:param>` tag is used to specify parameters that can be passed to the XSLT script being executed by the `<x:transform>` tag. The `<x:param>` tag should be used only inside the body of an `<x:transform>` tag.

```
// Syntax 1: Parameter value specified in attribute value
<x:param name="name" value="value"/>
// Syntax 2: Parameter value specified in the body content
<x:param name="name">
  parameter value
</x:param>
```

The `<x:param>` tag has the following attributes:

Attribute	Required	Purpose
<code>name</code>	Y	Specifies the name of this parameter.
<code>value</code>	N	Specifies the value of this parameter.

The <x:parse> Tag

The `<x:parse>` tag is used to parse an XML document. This tag must be called before any of the other XML tags can be used. Once the document has been parsed, it may be fed to the other XML tags for processing.

```
// Syntax 1: XML document specified via a String or Reader object
<x:parse {xml="XMLDocument"}
  [filter="filter"]
  [systemId="systemId"]
  [{var="varName" [{scope="{page|request|session|application}"]
  |varDom="varName"} [scopeDom="{page|request|session|application}"}]}]]/>
// Syntax 2: XML document specified via the body content
```

```

<x:parse [filter="filter"]
[systemId="systemId"]
[{var="varName" [{scope="{page|request|session|application}"}
scopeDom="{page|request|session|application}"]}]>
XML Document to parse
</x:parse>

```

The `<x:parse>` tag has the following attributes:

Attribute	Required	Purpose
<code>filter</code>	N	Allows a filter, of the type org.xml.sax.XMLFilter, to be used to filter the incoming XML data.
<code>scope</code>	N	Specifies the scope of the scoped variable referenced by the var attribute. Defaults to page.
<code>scopeDom</code>	N	Specifies the scope of the scoped variable referenced by the varDom attribute. Defaults to page.
<code>systemId</code>	N	Allows a system ID to be specified to return information about the parse.
<code>var</code>	N	Specifies the scoped variable to hold the parsed XML document.
<code>varDom</code>	N	Specifies the scoped variable that will receive the XML document as an org.w3c.dom.Document object.
<code>xml</code>	N	Specifies the XML that is to be parsed.

The `<x:set>` Tag

The `<x:set>` tag is used to set a scoped variable to one of the values found inside the XML document.

```

<x:set select="XpathExpression"
var="varName" [scope="{page|request|session|application}"]/>

```

The `<x:set>` tag has the following attributes:

Attribute	Required	Purpose
<code>scope</code>	N	Specifies the scope for the scoped variable referenced by the var attribute. The default is page.
<code>select</code>	Y	An XPath expression that you want to display.
<code>var</code>	Y	Specifies the scoped variable that is to be set.

The `<x:transform>` Tag

The `<x:transform>` tag is used to transform an XML document using the specified XSLT template. This allows you to use server-side XSLT in conjunction with your JSTL Web applications.

```

// Syntax 1: No body content
<x:transform
xml="XMLDocument" xslt="XSLTStylesheet"
[xmlSystemId="XMLSystemId"] [xsltSystemId="XSLTSystemId"]
[{var="varName" [scope="scopeName"]|result="resultObject"}]
// Syntax 2: Transformation parameters specified using body
<x:transform
xml="XMLDocument" xslt="XSLTStylesheet"
[xmlSystemId="XMLSystemId"] [xsltSystemId="XSLTSystemId"]
[{var="varName" [scope="scopeName"]|result="resultObject"}]
<x:param> actions
</x:transform>
// Syntax 3: Using a body to both specify an XML document and
transformation parameters
<x:transform
xslt="XSLTStylesheet"
xmlSystemId="XMLSystemId" xsltSystemId="XSLTSystemId"
[{var="varName" [scope="scopeName"]|result="resultObject"}]
XML document being processed
optional <x:param> actions
</x:parse>

```

The `<x:transform>` tag has the following attributes:

Attribute	Required	Purpose
<code>result</code>	N	A scoped variable of type <code>javax.xml.transform.Result</code> that specifies how the result should be processed.
<code>scope</code>	N	Specifies the scope of the scoped variable referenced by the <code>var</code> attribute. Defaults to page.
<code>var</code>	N	Specifies a scoped variable that will hold the transformed document. If none is specified, the results will be written to the page.
<code>xml</code>	N	Specifies a string that holds the XML file to be used with the transformation.
<code>xslt</code>	Y	Specifies a string that holds the XSLT file to be used with the transformation.

The `<x:when>` Tag

The `<x:when>` tag should be used inside the body of an `<x:choose>` tag. The first `<x:when>` tag that is satisfied will execute any tags in its body. If no `<x:when>` tags are satisfied, then the `<x:otherwise>` tag will be executed.

```

<x:when select="XPathExpression">
body content
</x:when>

```

The `<x:when>` tag has one attribute:

Attribute	Required	Purpose
-----------	----------	---------

<code>select</code>	Y	An XPath expression that you want to evaluate.
---------------------	---	--

Appendix B. Installing JSTL and Tomcat

In this appendix, we show you how to install Tomcat and JSTL for use with this book. We focus on the Windows installation, but you can use these instructions with other operating systems as well. For specific information on how to install JSTL in a Unix environment, refer to [Appendix D](#), "Unix Installation Notes."

To compile and execute the examples in this book, you must install several components: a JDK version, Tomcat, and JSTL. Once these three components are installed, you are ready to develop JSTL Web applications and use the examples in this book.

Some of our examples also use a database connection. To use these examples, you must also install a database, such as MySQL. We cover installing MySQL in [Appendix C](#).

Installing JDK

To run any of the examples in this book, you have to install the JDK. We tested our examples with JDK 1.4, so we suggest you use JDK 1.4 or higher for JSTL.

You can download the JDK from the Sun Web site at <http://java.sun.com/j2se>. Make sure that you download the full SDK, not just a runtime environment (JRE). The file you download will be named `j2sdk-1_4_0_01-windows-i586.exe`, or something similar. You should execute this file to begin the process of installing the JDK.

Executing the downloaded file will launch an Install Shield installation program. This program makes the process of installing the JDK relatively easy. Once you've installed the JDK, the installation program exits and you are ready to install Tomcat.

Installing Tomcat

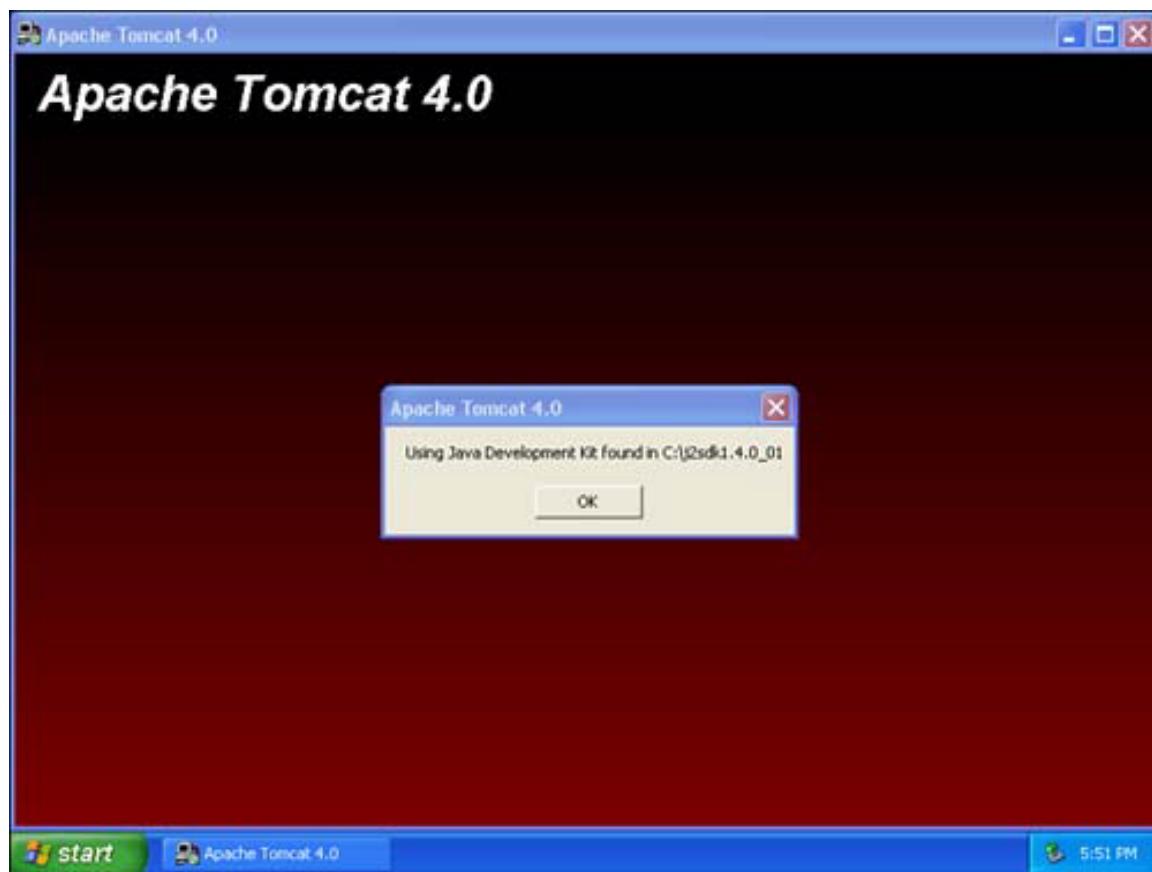
JSTL requires JSP pages to execute. To execute JSP pages, you must have a JSP-compatible Web server installed on your system. JSTL should work with any JSP 1.2 or higher Web server. The Web server we use in this book is the Apache Tomcat Web server. In this section, we show you how to install Tomcat.

The examples in this book were tested with Apache Tomcat v4.0.3. We recommend that you use at least this version of Tomcat. Before you install Tomcat, it is absolutely essential that you have the JDK installed. Refer to the previous section for more information on installing the JDK.

You can download the Tomcat Web server from the Apache Web site at <http://jakarta.apache.org/tomcat/>. When you begin the download, you'll notice that several versions are available. If you are using Windows, you can choose between a light (LE) and a full version. Both versions are meant to function exactly the same. The only difference is that the LE version does not include some of the JAR files normally included with JDK 1.4, such as for XML processing. The result is that if you are using JDK 1.4, you can choose the LE version and not have to download JAR files that you already have. The safest route is usually just to install the full (non-LE) version.

Once you download Tomcat, you should have a file named `Jakarta-tomcat-4.0.3.exe`. This file, when executed, functions as the installer for the Tomcat Web server. To begin the installation, execute `Jakarta-tomcat-4.0.3.exe`. Once the Tomcat installation program begins, you see a window that looks like the one shown in [Figure B.1](#).

Figure B.1. Installing Apache Tomcat.



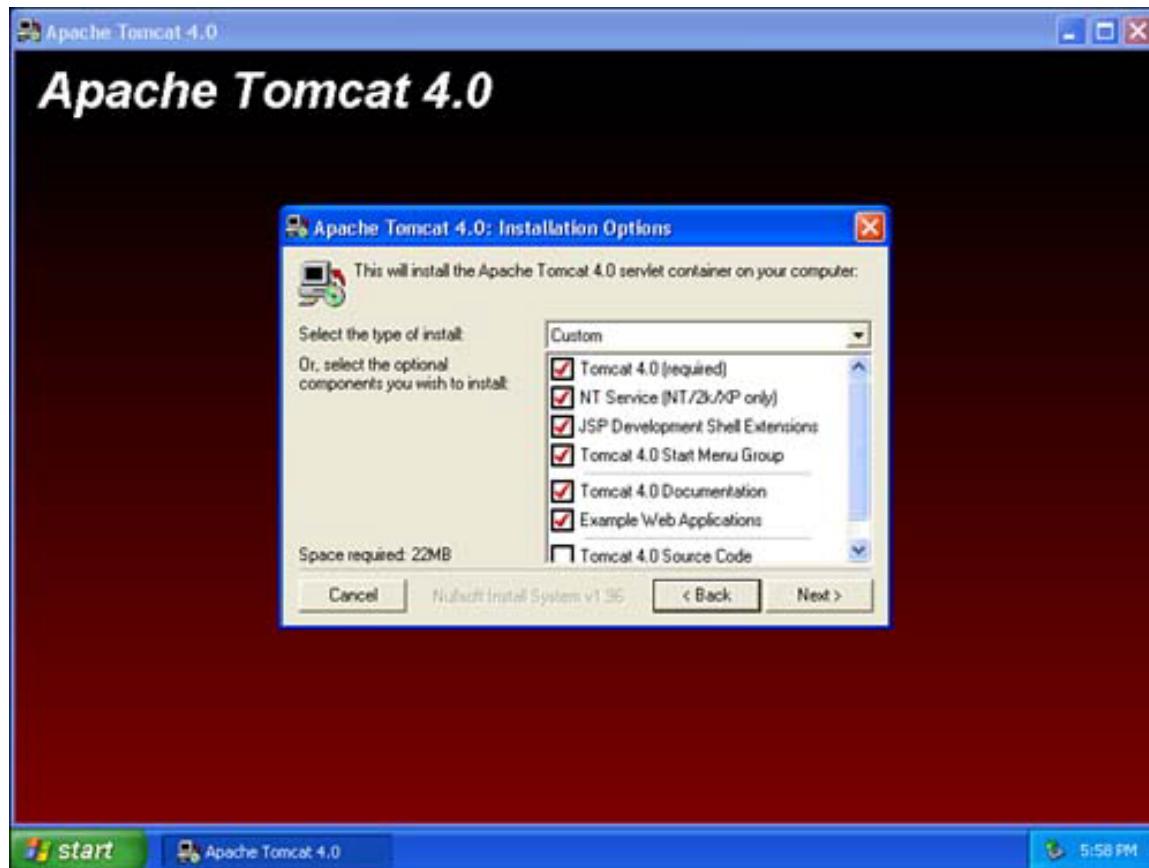
The first thing that the installation program will do is attempt to locate your JDK installed path. If the program finds a valid JDK, it displays the path, as shown in [Figure B.1](#). If the program does not find a valid JDK, it displays an error.

The installation program will guide you through the process of installing Tomcat. One option that the program provides is the ability to install Tomcat as a service. If you are

using Windows XP, 2000 Pro, or Windows NT, you should install Tomcat as a service. Doing this will cause Tomcat to run in the background of your computer, even when no user is logged into the computer.

Installing Tomcat as a service requires that you select this as one of the options while the Tomcat installation program is running. The window that allows you to specify to run Tomcat as a service is shown in [Figure B.2](#).

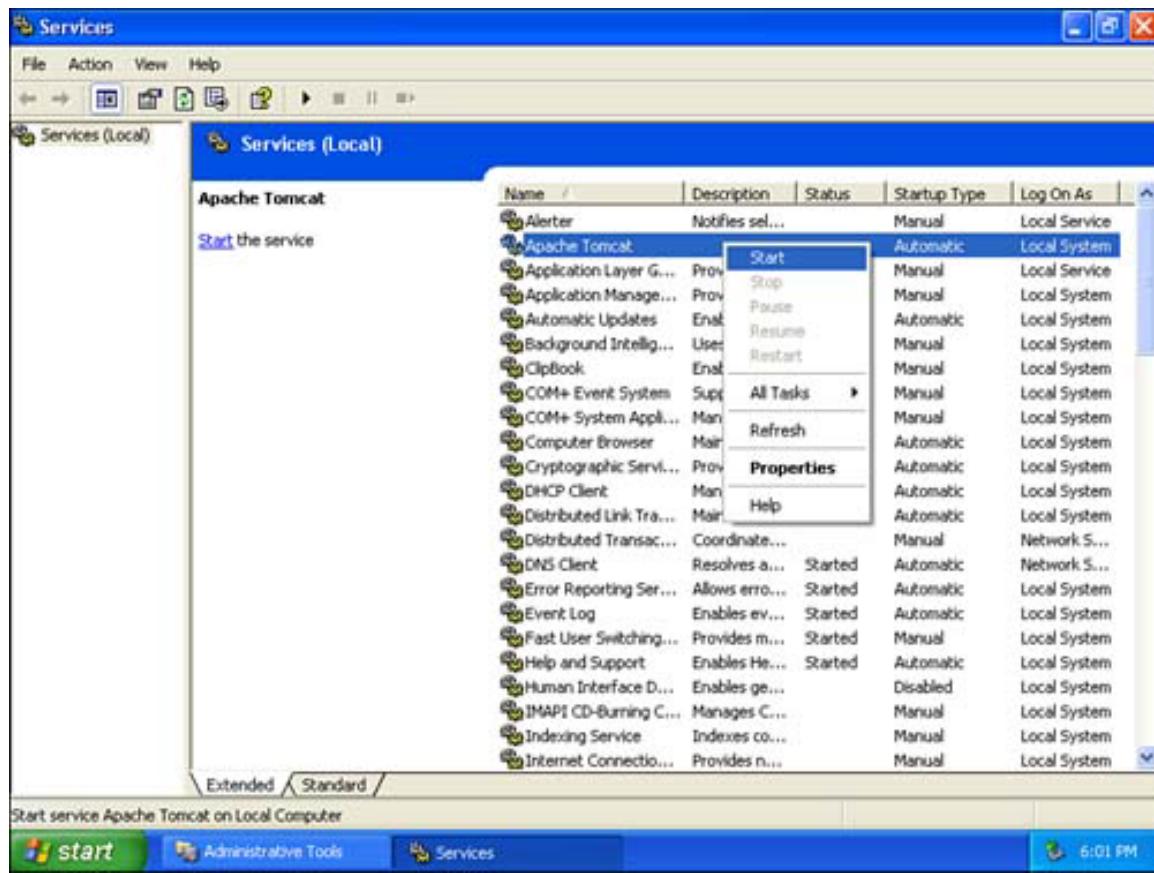
Figure B.2. Tomcat install options.



If you choose to run Tomcat as a service, you must use the services manager to start Tomcat. From the services control panel, you are able to see all of the services that your computer is currently running or is aware of. If you look at the line for the Tomcat Web server, you'll notice that it is set to automatically start, yet it is not running. This means that Tomcat will be started the next time you restart your computer.

While you could restart your computer now to start Tomcat, it is preferable to use the Services window to start Tomcat. To do so, right-click the Tomcat line and choose Start. Once you do, Tomcat will start. [Figure B.3](#) shows this process.

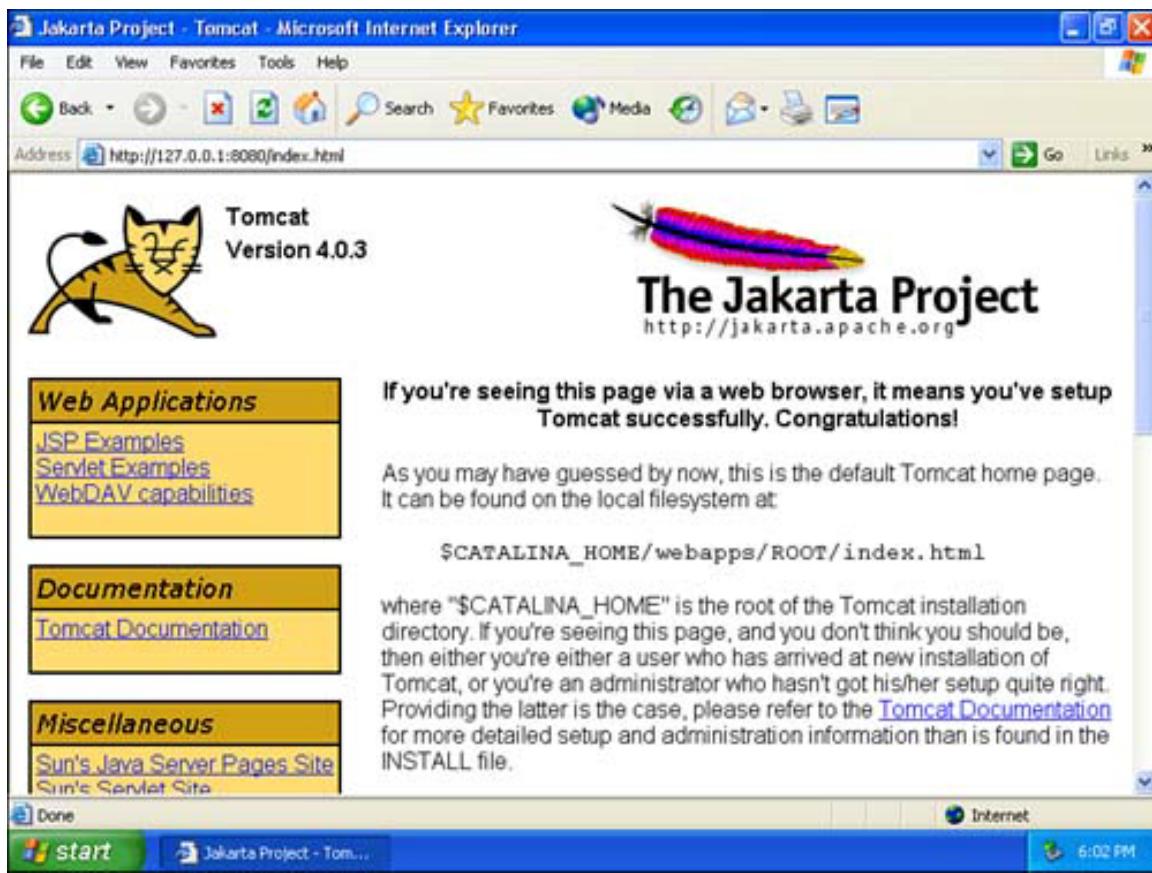
Figure B.3. Starting Tomcat.



The Services window allows you to restart Tomcat when needed. This is a handy feature because you will frequently need to restart Tomcat. Any time that you deploy new JAR files to Tomcat, for example, you'll have to restart the Tomcat service. If Tomcat is not installed as a service, the installation program places shortcuts for starting and stopping Tomcat in the Programs menu.

Now that Tomcat is installed and running, we can test it. To test Tomcat, open a browser on the computer that Tomcat is running on. Once this browser starts, enter the Web address <http://localhost:8080/>. This Web address takes you to the Web server that is running on your local computer on port 8080. If localhost does not work, you can try entering the Web address <http://127.0.0.1:8080/>. If all works well, your screen should resemble the one shown in [Figure B.4](#).

Figure B.4. Accessing Tomcat.



By default, Tomcat will use port 8080 as the Web address. If you would like to set up Tomcat so that it runs on port 80, which is the usual Web port, you must modify Tomcat's server.xml file. Once you modify Tomcat to run on port 80, you will no longer need to specify a port address. If Tomcat is running on port 80, it is enough to use the Web address `http://localhost` alone to access the Web server

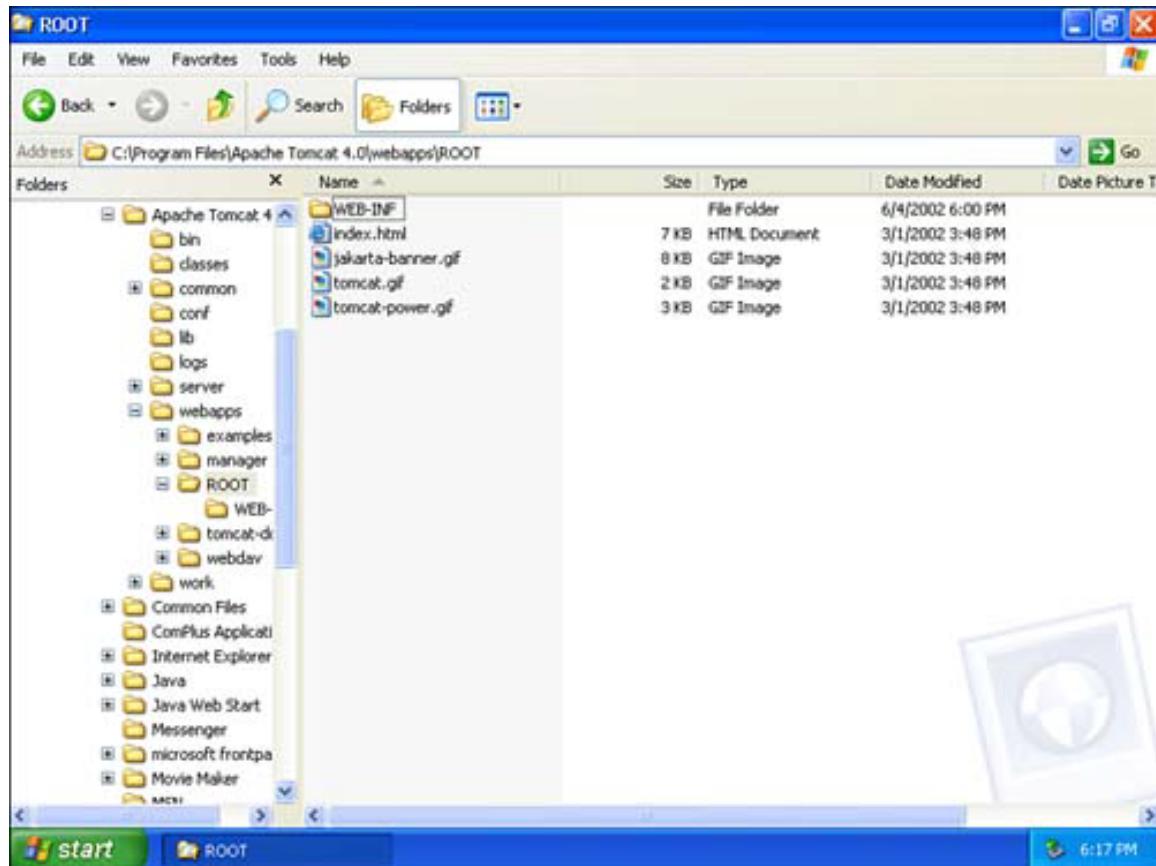
Installing the Book Examples

Now that you have Tomcat properly installed, you are ready to install the sample programs from this book. You can download these programs from <http://www.sampspublishing.com/>. The examples are packaged as one Web directory that you can copy to the Tomcat Web server. This directory contains both the examples and JSTL 1.0. This is the easiest way to install the examples. However, there are times when you may want to directly install each component. Later in this appendix, we explain how to install JSTL without the book examples.

Tomcat stores all of the Web files it needs in a directory named webapps. When you download the examples that come with this book, you'll notice that the Zip archive includes a directory named webapps, which contains all of the book examples. Simply

overwrite your Tomcat Web server's webapps directory with this one from Sams. [Figure B.5](#) shows the directory structure.

Figure B.5. The Tomcat directories.

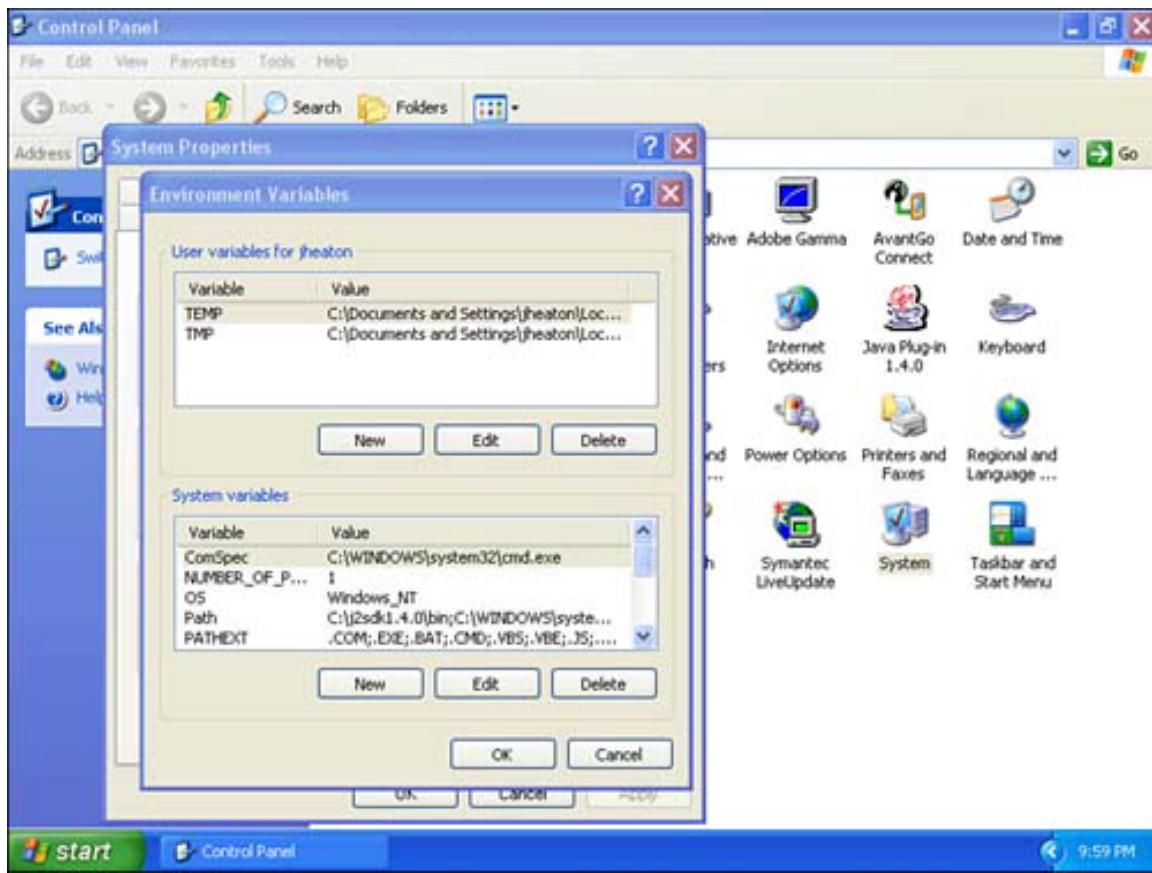


The Classpath and Search Path

It is usually a good idea to make the BIN directory for the JDK a part of the system search path. Doing this will allow you to type the command javac from any directory and be able to compile a program. You also have to modify your classpath so that certain Tomcat JAR files are available to the Java compiler. You must do this in order to execute the examples from [Chapter 11](#).

Both the classpath and system search path are defined in the same area. If you are using Windows XP, Windows 2000 Pro, or Windows NT, you set them in the System window in Control Panel, as shown in [Figure B.6](#).

Figure B.6. The System window.



You have to modify the system path to include the path of your JDK's **BIN** directory. The system path is stored in the environment variable named **PATH**. Your JDK **BIN** directory is most likely **C:\j2sdk1.4.0\bin** or something similar. Add this path to your **PATH** environment variable.

In addition, you have to add a few JAR files to your classpath. This allows the Java compiler to correctly compile the tag library that we create in [Chapter 11](#). If you do not want to modify your classpath, you can modify the [Chapter 11](#) script files to include the required classpath, as shown here:

```
C:\Program Files\Apache Tomcat 4.0\webapps\ROOT\WEB-INF\lib\mm.mysql-  
2.0.12-bin.jar  
C:\Program Files\Apache Tomcat 4.0\common\lib\servlet.jar  
C:\Program Files\Apache Tomcat 4.0\webapps\ROOT\WEB-INF\lib\jstl.jar  
C:\Program Files\Apache Tomcat 4.0\webapps\ROOT\WEB-  
INF\lib\standard.jar
```

The first file is the MM driver that allows us to access the MySQL database. The second file, servlet.jar, allows us to access the servlet context variables. The last two lines enable us to access the JSTL tag library; this is important because the tag library that we create in [Chapter 11](#) is designed to work with the tag library.

Installing JSTL without the Examples

If you follow the instructions in the previous section, you will correctly install JSTL and the book examples. In this section, we explain how to install JSTL apart from the book's examples. If you want to install JSTL into a server environment, this is the procedure you follow.

The exact directory that JSTL will install into varies somewhat, depending on the version. In the 1.0 version, there is a `jstl-1.0` directory, and in 1.0.1 it is called `standard-1.0.1`. Inside this directory is a `lib` directory that contains several files. [Table B.1](#) documents the purpose of each of these files.

Table B.1. The JSTL Files	
File	Purpose
<code>c.tld</code>	The core tag library (EL version)
<code>c-rt.tld</code>	The core tag library (RT version)
<code>dom.jar</code>	
<code>fmt.tld</code>	The I18N tag library (EL version)
<code>fmt-rt.tld</code>	The I18N tag library (RT version)
<code>jaxp-api.jar</code>	Required JAR file for JSTL
<code>jaxen-full.jar</code>	Required JAR file for JSTL
<code>jdbc2_0-stdext.jar</code>	Required JAR file for JSTL
<code>jstl.jar</code>	Required JAR file for JSTL
<code>sax.jar</code>	Required JAR file for JSTL
<code>saxpath.jar</code>	Required JAR file for JSTL
<code>sql.tld</code>	Database tag library (EL version)
<code>sql-rt.tld</code>	Database tag library (RT version)
<code>standard.jar</code>	Required JAR file for JSTL
<code>xalan.jar</code>	Required JAR file for JSTL
<code>xerxesImpl.jar</code>	Required JAR file for JSTL
<code>x.tld</code>	XML tag library (EL version)
<code>x-rt.tld</code>	XML tag library (RT version)

To properly install JSTL for your Web application, you must copy these files to specific locations. The two JAR files (`standard.jar` and `jstl.jar`) must be copied to your Webapp's library directory. This ensures that these files will be part of the classpath. Under the Windows operating system, your lib directory for Tomcat will be `C:\Program Files\Apache Tomcat 4.0\webapps\ROOT\WEB-INF\lib`.

You copy the tag definition files, which have a `.tld` extension, to your `WEB-INF` directory. This is typically a child directory of the Web root directory for your Web application. The Web root directory is the directory that holds all your JSP and HTML files. Under

the Windows operating system, Tomcat's Web root directory is typically `C:\Program Files\Apache Tomcat 4.0\webapps\ROOT`. Tomcat's `WEB-INF` directory is typically `C:\Program Files\Apache Tomcat 4.0\webapps\ROOT\WEB-INF`.

The last step is to let the Web server know where the TLD files are stored. You do this by modifying the `web.xml` file, which should be located in your `WEB-INF` directory. Listing B.1 shows a `web.xml` file that uses all four JSTL tag libraries.

Listing B.1 A web.xml That Uses All Four JSTL Taglibs

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
    <taglib>
        <taglib-uri>http://jakarta.apache.org/taglibs/
core</taglib-uri>
        <taglib-location>/WEB-INF/c.tld</taglib-location>
    </taglib>

    <taglib>
        <taglib-uri>http://jakarta.apache.org/taglibs/sql</taglib-uri>
        <taglib-location>/WEB-INF/sql.tld</taglib-location>
    </taglib>

    <taglib>
        <taglib-uri>http://jakarta.apache.org/taglibs/fmt</taglib-uri>
        <taglib-location>/WEB-INF/fmt.tld</taglib-location>
    </taglib>

    <taglib>
        <taglib-uri>http://jakarta.apache.org/taglibs/xml</taglib-uri>
        <taglib-location>/WEB-INF/x.tld</taglib-location>
    </taglib>

</web-app>
```

As you can see in Listing B.1, each of the four tag libraries has a separate taglib entry. This is the same format that is used for every tag library you'll use. To use any tag library, you simply include the correct taglib-uri and taglib-location properties. Now that you have successfully installed the JSTL required files to their proper locations, you can construct a JSP page that contains JSTL tags.

Appendix C. Installing MySQL

To create useful Web applications, you must use some sort of a database. In this appendix, we show you how to install a MySQL database. MySQL is a popular database that is released under a GNU Public License (GPL). There is no charge for using MySQL.

Some of the applications we present in this book require that you use a relational database. Several chapters use a forum application that demonstrates JSTL features. The forum application requires a database to maintain its user lists and message areas. In this appendix, you'll learn how to create a MySQL database that is ready to work with the forum applications we present in this book.

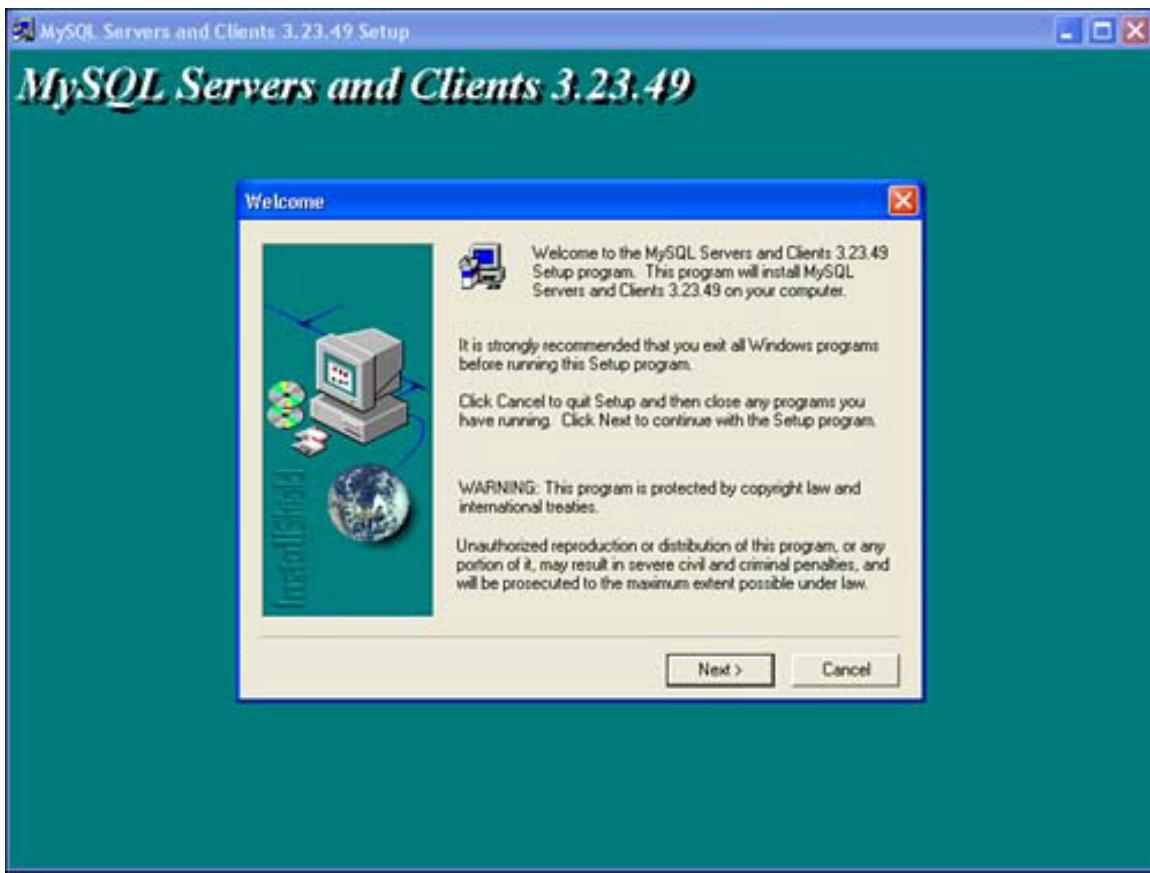
Obtaining and Installing MySQL

You must download a copy of MySQL to install it to your computer system. Prebuilt binary versions of MySQL are available for all popular operating systems. To download a copy of MySQL, go to <http://www.mysql.com/>. From this Web site, you can access a download section that allows you to choose among several versions of MySQL.

Notice that there are two main versions of MySQL. A full-featured version, named MySQL-Max, is the one you should download. The regular version of MySQL has no support for some of the more advanced features. For this book, we assume that you are using the Max version; we used MySQLMax 3.23.

Once you download the Windows version of MySQL, you should have a file named something like `mysql-3.23.48-win.zip`. (This is a zipped file. If you are unable to unzip files, refer to the site <http://www.winzip.com/>.) Once you unzip the MySQL file, you'll find a standard installation program. Run the program named setup to install MySQL to your system. This process is shown in [Figure C.1](#).

Figure C.1. Installing MySQL.



Now that MySQL is installed, you must expose its `BIN` directory to the system path. Doing this allows you to execute MySQL commands from any directory at a command prompt. By default, `c:\mysql\bin` is the directory where MySQL is installed.

The procedure for adding the MySQL `BIN` directory to your system path differs, depending on which operating system you are running. If you're using Windows XP, Windows 2000 Pro, or Windows NT, you set the system path using the System Properties window in Control Panel, shown in [Figure C.2](#).

Figure C.2. Setting environment variables.

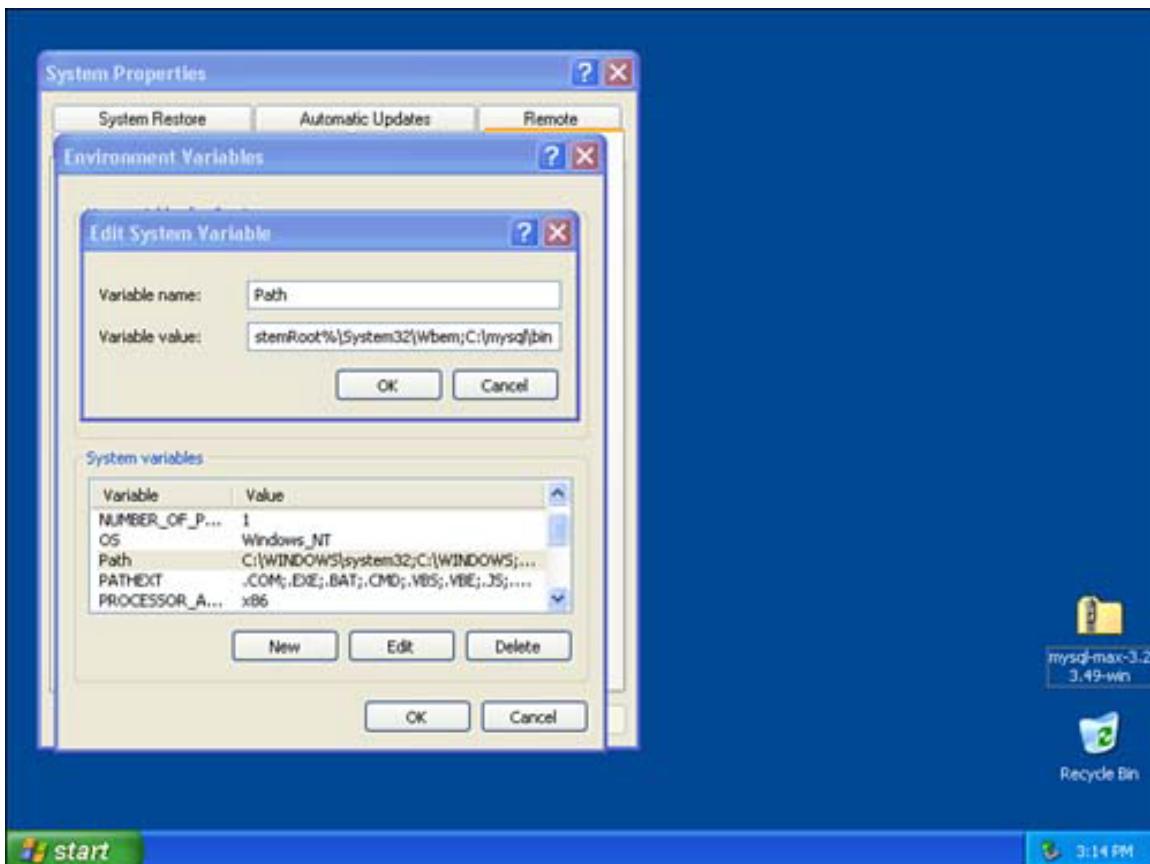


Figure C.2 shows the System window for Microsoft Windows XP. Although this window will look slightly different in older versions, the procedure is the same. You must add the `c:\mysql\bin` directory to the end of your `PATH` environment variable. If there is already a path in your `PATH` environment variable, you must separate the new path with a semicolon (;), as shown in Figure C.2.

If you are using an older version of Windows, such as Windows 95, Windows 98, or Windows ME, you must modify your autoexec.bat file in order to change the system path. In your autoexec.bat file, you should see a PATH statement. The exact layout of your autoexec.bat file depends on what other software has already been installed on your computer system. Once you locate the PATH statement, you must append the path `c:\mysql\bin\` to the end of the specified PATH statement. Of course, you should use a semicolon (;) to separate the new path.

When you installed the Java JDK, you also modified the system path. It is important that you do not change the JDK path information that you set up before. Make sure that you just append the MySQL path information to the end of what is already there.

If you've followed these steps properly, you now have the MySQL software installed on your system. Now you have to set up the database so that it is ready for use. In the next section, we look at that process.

Setting Up MySQL

Now that you've installed your MySQL database system, you must be able to launch your server. If you're using Windows XP, Windows 2000 Pro, or Windows NT, we suggest that you configure MySQL to run as a service. If you're using an older version of Windows, refer to the MySQL documentation.

If you're using Unix, you do not need to worry about services. MySQL for Unix installs to be ran as a daemon process. For more information about using Unix, refer to [Appendix D](#), "Unix Installation Notes."

By configuring MySQL to work as a service, you are guaranteed that the MySQL database system will always be running. Services run even when no user is currently logged into the system. To set MySQL to run as a service, you have to start a command prompt window. From this window, change the current directory to the `BIN` directory of MySQL.

Earlier, we set up the path so that you can execute the MySQL commands from anywhere. When you're installing the service, it is important that you do not install these commands from anywhere. To install MySQL to run as a service, you must change to the `c:\mysql\bin` directory. To do this, issue the following command in the command prompt window:

```
cd \mysql\bin
```

Now that you are in the `BIN` directory, you have to instruct MySQL to install itself as a service:

```
mysql-max-nt --install
```

At this point, MySQL is set up to run as a service. The next time that you restart your computer, MySQL will run in the background and you can easily access it. To start the service without having to restart your computer, use this command:

```
net start mysql
```

Now that the service is running, you can connect to your MySQL database for the first time. The following mysql command connects you to the MySQL instance you just installed (as the root user):

```
mysql -u root
```

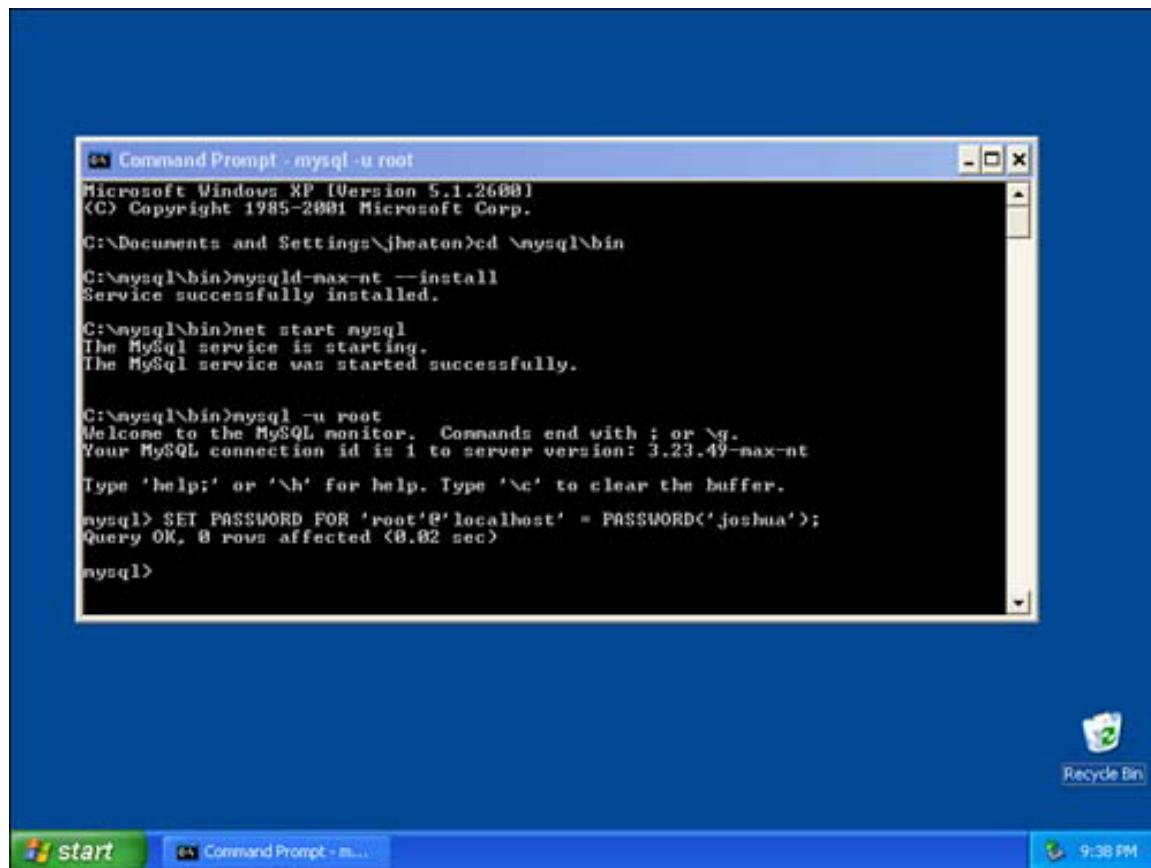
Notice that you are allowed to access the system with no password. This is necessary for the first-time install of MySQL. We suggest that you now change your password for security reasons. For example, this command sets the root user's password to joshua:

```
SET PASSWORD FOR 'root'@'localhost' = PASSWORD('joshua');
```

If you followed each of these steps correctly, your screen should look like the one in [Figure C.3](#). You can now exit MySQL with the following command:

```
exit
```

Figure C.3. Setting the MySQL service.



Now you must enter MySQL with the root user and specify a password. Use the following command:

```
mysql -u root -p
```

You are not allowed to specify your password as part of the command line because of security reasons. Other users can easily view a list of running processes for your account. This list shows all command-line parameters that you used, so if you specify your password on the command line, other users could see it. Instead, specifying the `-p` parameter tells MySQL that you are going to log in using a password.

MySQL will now prompt you for a password. Enter joshua (or whatever you choose as your password).

Now that the root user has been properly set up, let's create the forum database that we use for the examples in this book. First, issue this command:

```
create database forum;
```

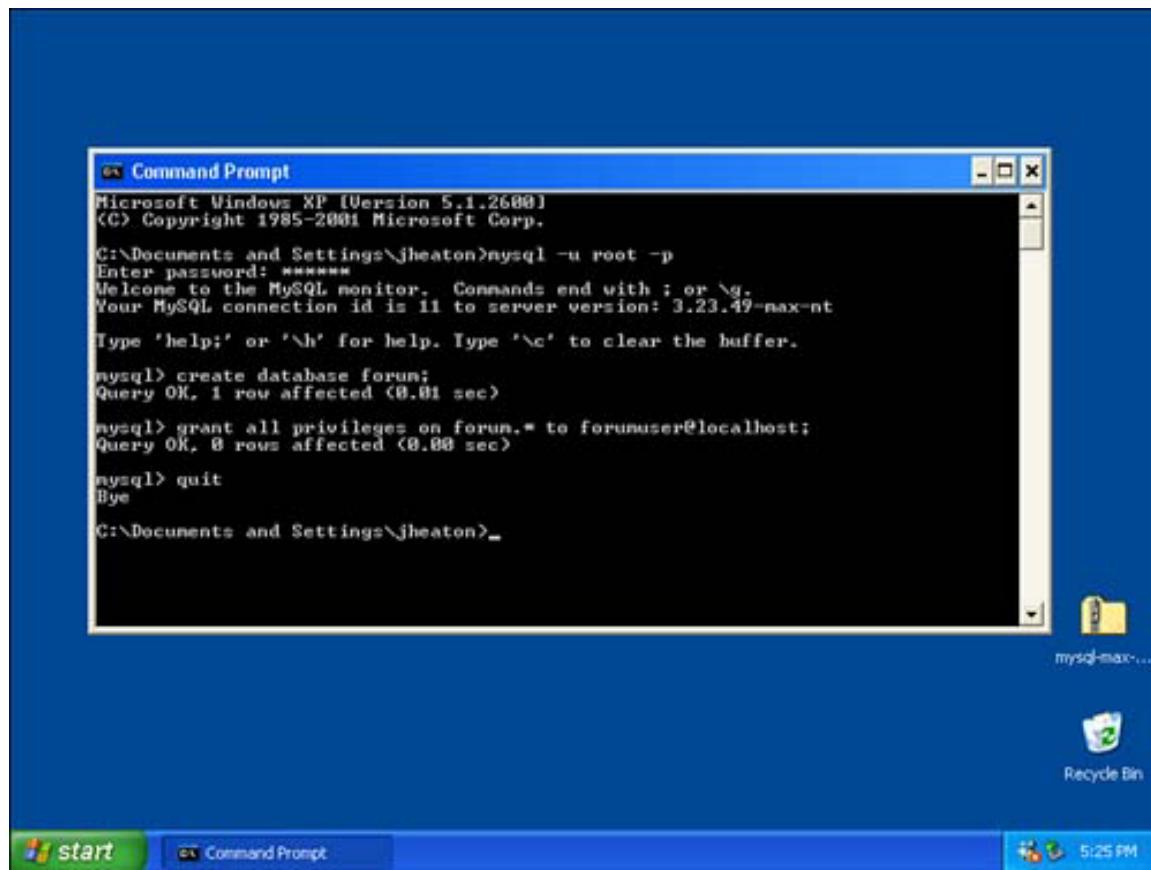
Once you execute this command, you'll have a new database named forum. Now you have to create a user to access the forum database. We suggest you always create users specific to your Web applications. This is more secure than using the root user to access each of these databases. To create a user, you must grant the user rights.

Granting a user rights for the first time also creates that user:

```
grant all privileges on forum.* to forumuser@localhost
```

You now have a user named forumuser that can access your forum database. If you've followed each step correctly, your screen should now look like the one in [Figure C.4](#).

Figure C.4. Logging in using a password.



One very important thing to note about this command is that it ties the server name to the user. If you'd like to access your database from another computer, it is necessary to also create a user account for that machine. For example, if you want to access the forum

database from a computer with the IP address of 192.168.1.100, issue the following command:

```
grant all privileges on forum.* to forumuser@192.168.1.100
```

You have now successfully installed MySQL and created a database to hold the forum application. In the next section, you'll learn how to create tables inside the forum database.

Creating the Forum Example

Now that you've installed MySQL and created a forum database, you must install the database. You do this using a script file provided at the Sams Web site. This script file, which is explained in greater detail in [Chapter 7](#), contains all of the SQL necessary to create your forum database. The script is designed to be called against either a blank database or a database that already has a forum. If you run this script against an existing forum database, it will reset the database back to its original state.

To run this script, you must be at a command prompt. Make sure you are in a directory that contains `createforum.sql`. From the command prompt, issue this command:

```
mysql -u forumuser <createforum.sql
```

Once this command executes, you will have the tables necessary to test the forum application. To ensure that you installed everything correctly, you should perform a query against the `t_users` table. To do this, enter MySQL with the following command:

```
mysql -u forumuser
```

Now that you are inside MySQL, you must set the correct database:

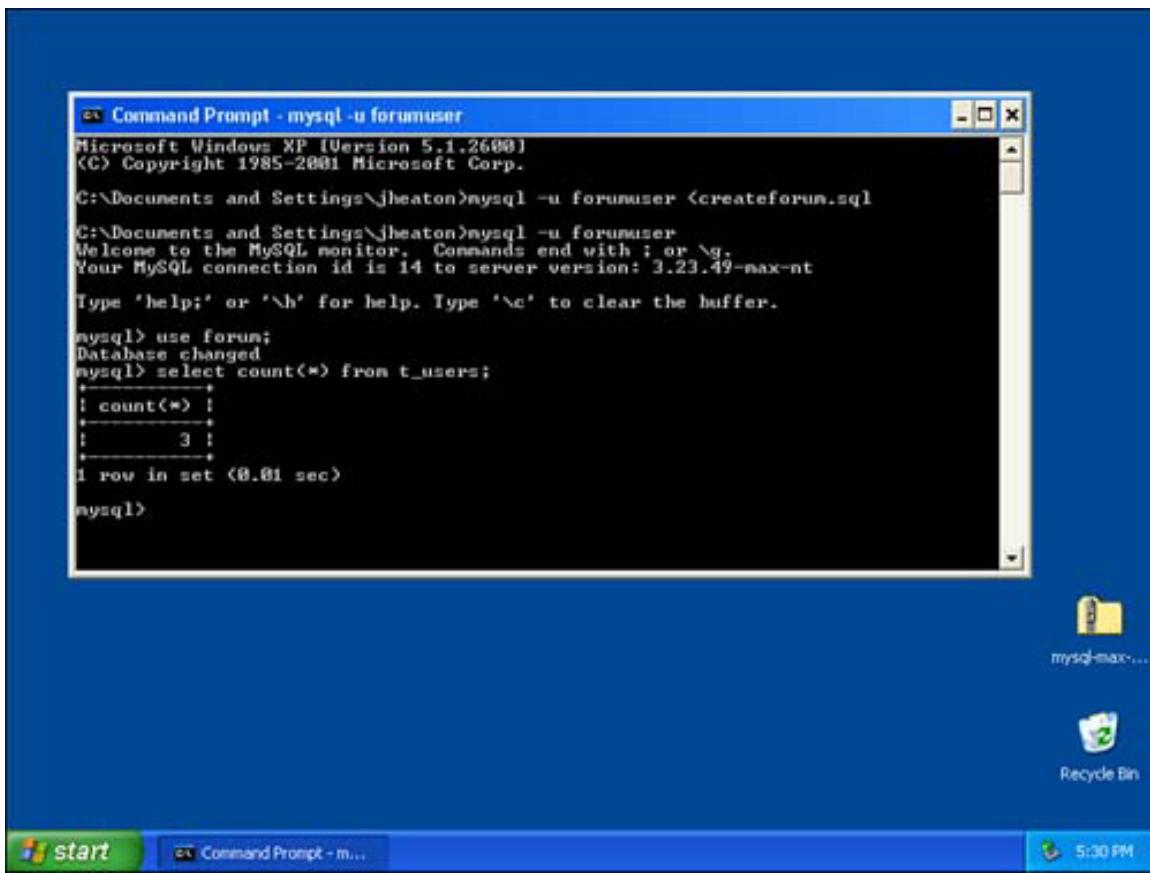
```
use forum;
```

At this point, you can perform a basic SQL SELECT statement, such as the following:

```
select * from t_users;
```

If all of these commands execute properly, your screen should look like the one shown in [Figure C.5](#).

Figure C.5. Setting up the forum database.



You now have MySQL properly installed and the forum database ready for use. One configuration issue remains: You must install a driver so that Java can get to MySQL.

Installing a JDBC Driver

For Java to use a database, you have to have a JDBC driver. Several drivers are available that you can use. The driver that we use in this book is the MM driver, which you can download from <http://mymysql.sourceforge.net/>. If you followed the instructions in [Appendix B](#), you won't need to install the MM driver. The examples download contains the MM driver already properly installed.

The MM driver is distributed as a JAR file named `mm.mysql-2.0.14-you-must-unjar-me.jar`. This is not an ordinary JAR, and it cannot be used directly by Java. The author of MM is simply using JAR as an archive format, similar to Zip. The easiest way to extract from this archive is to rename it to a `.zip` file and use a Zip tool.

Once you access the contents of this archive, locate the MM JAR file, which should be named `mm.mysql-2.0.12-bin.jar` or something similar. You must make this file

accessible to the classpath. For the purposes of this book, this involves placing the file in the directory `C:\Program Files\Apache Tomcat 4.0\webapps\ROOT\WEB-INF\lib`.

Now that you have the MM driver installed, you are ready to use MySQL from JDBC. To access any driver from JDBC, you must know the driver name and the URL of the database. The driver name for the MM driver is `org.gjt.mm.mysql.Driver`, and the URL is `jdbc:mysql://localhost/forum?user=forumuser`. To use this information in JSTL, use the following `<sql:setDataSource>` tag:

```
<sql:setDataSource var="dataSource" driver="org.gjt.mm.mysql.Driver"
url="jdbc:mysql://localhost/forum?user=forumuser"
scope="session" />
```

See [Chapters 7](#) and [11](#) for more information about using databases from JDBC.

Appendix D. Unix Installation Notes

This book contains many examples that illustrate the use of JSTL. [Appendices B and C](#) explain how to set up your environment for the Windows platform. The instructions in these two appendixes generally apply to using JSTL with Unix as well.

We developed our examples on a system that runs Red Hat Linux 7.2. This was our primary development environment, so all of the code in this book is well tested under Linux.

Linux versions are available for Tomcat, MySQL, and the JDK. You should install all of these programs to their standard locations. You must also modify your profile file so that the classpath and system path are set up properly.

Under Unix, the procedure for changing your profile file varies greatly, depending on which shell you are running. [Listing D.1](#) shows a simple profile for the BASH shell, which is the default shell for Red Hat Linux.

[Listing D.1 A Sample .bash_profile File](#)

```
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH:$HOME/bin:/usr/java/j2sdk1.4.0/bin
JAVA_HOME=/usr/java/j2sdk1.4.0/
CLASSPATH=./usr/jar/mm.mysql-2.0.12-bin.jar:/var/tomcat4/common/lib/
    servlet.jar:/var/tomcat4/webapps/ROOT/WEB-INF/lib/jstl.jar:/var/
    tomcat4/webapps/ROOT/WEB-INF/lib/standard.jar

export PATH
export JAVA_HOME
export CLASSPATH
unset USERNAME
```

It is unlikely that you will want to copy this file directly over your existing profile. We used the script file in [Listing D.1](#) with a test user we created to test examples for this book. Regardless of how much of this file you use, it does show you where the various paths are likely to be set up. If your system installs these components to different paths, you must use your own paths.