murach's

Java servlets and JSP

(Chapter 4)

Andrea Steelman
Joel Murach

Mike Murach & Associates

2560 West Shaw Lane, Suite 101
Fresno, CA 93711-2765
(559) 440-9071 • (800) 221-5528

murachbooks@murach.com • *www.murach.com*

Section 2

The essence of servlet and JSP programming

The best way to learn how to develop web applications in Java is to start developing them. That's why the chapters in this section take a hands-on approach to developing web applications. In chapter 4, you'll learn how to code JavaServer Pages (JSPs), which is one way to develop web applications. In chapter 5, you'll learn how to code servlets, which is another way to develop web applications. And in chapter 6, you'll learn how to structure your web applications so you combine the best features of JSPs and servlets.

With that as background, you'll be ready to learn the other web programming essentials. In chapter 7, you'll learn how to work with sessions and cookies so your application can keep track of its users. In chapter 8, you'll learn how to build better applications by using JavaBeans. And in chapter 9, you'll learn how to simplify the code in your JSPs by using custom tags. When you complete this section, you'll have the essential skills that you need for designing, coding, and testing web applications that use JSPs and servlets.

4

How to develop JavaServer Pages

In this chapter, you'll learn how to develop a web application that consists of HTML pages and JavaServer Pages (JSPs). As you will see, JSPs work fine as long as the amount of processing that's required for each page is limited. When you complete this chapter, you should be able to use JSPs to develop simple web applications of your own.

The Email List application

This topic introduces you to a simple web application that consists of one HTML page and one *JavaServer Page*, or *JSP*. Once you get the general idea of how this application works, you'll be ready to learn the specific skills that you need for developing JSPs.
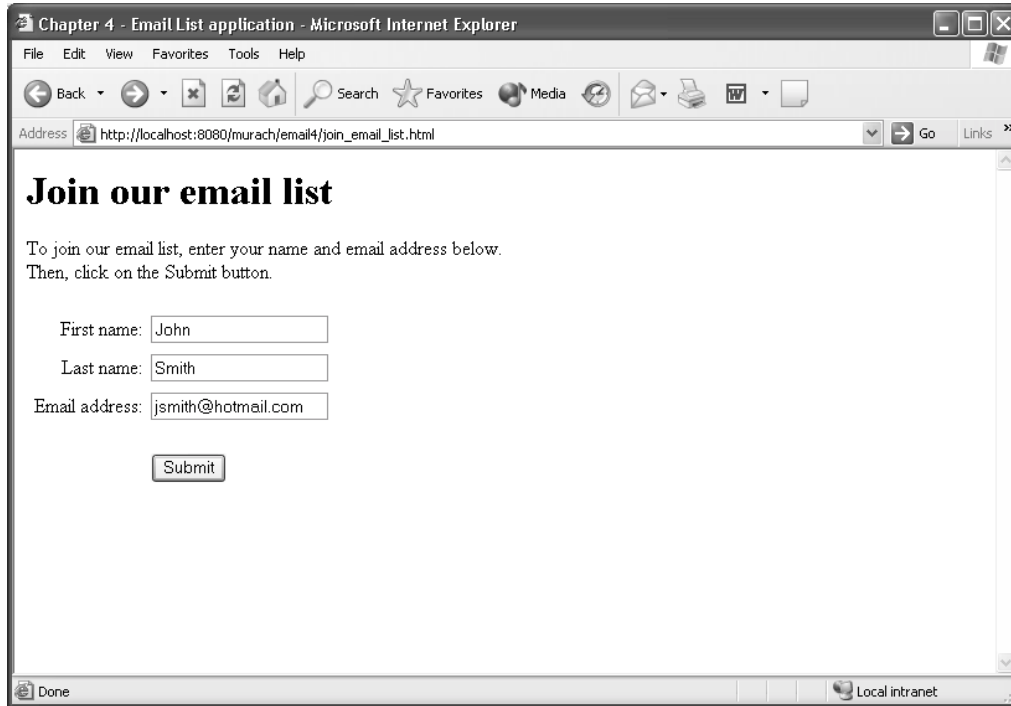
The user interface for the application

Figure 4-1 shows the two pages that make up the user interface for the Email List application. The first page is an HTML page that asks the user to enter a first name, last name, and email address. Then, when the user clicks on the Submit button, the HTML page calls the JSP and passes the three user entries to that page.

When the JSP receives the three entries, it could process them by checking them for validity, writing them to a file or database, and so on. In this simple application, though, the JSP just passes the three entries back to the browser so it can display the second page of this application. From this page, the user can return to the first page by clicking the Back button in the web browser or by clicking the Return button that's displayed on this page.

As simple as this application is, you're going to learn a lot from it. In this chapter, you'll learn how enhance this application so it uses regular Java classes to save the user entries in a text file. Then, in later chapters, you'll learn how to modify this application to illustrate other essential skills that apply to servlet and JSP programming.

The HTML page

The JSP
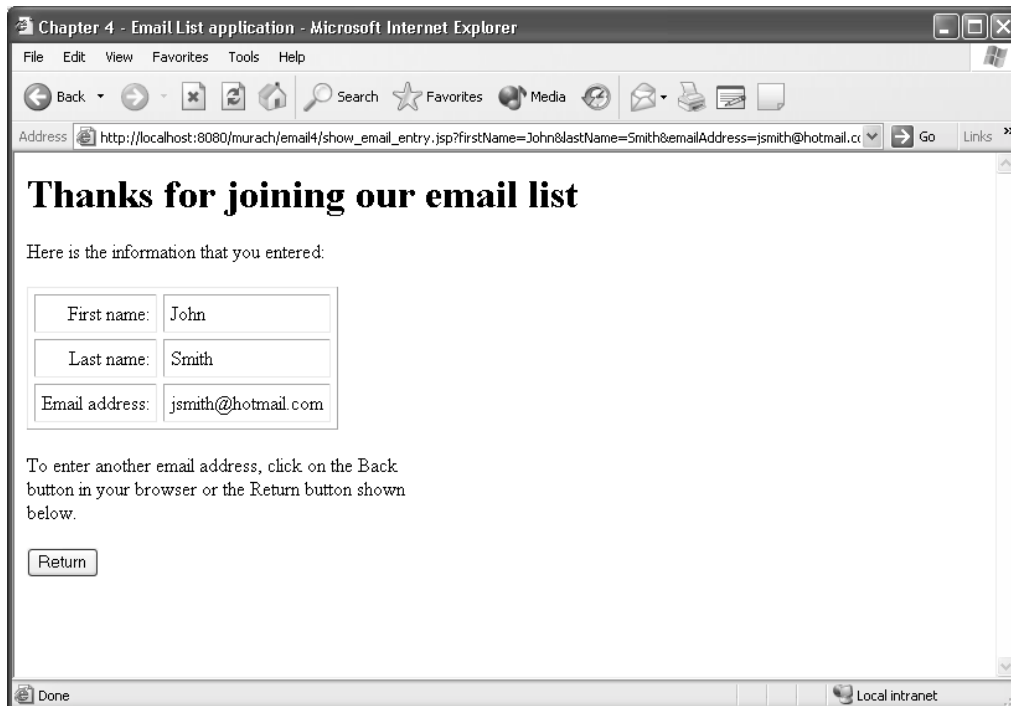
Figure 4-1   The user interface for the application

The code for the HTML page that calls the JSP

Figure 4-2 presents the code for the HTML page that calls the JSP. If you've read chapter 3, you shouldn't have any trouble following it. Here, the Action attribute of the Form tag calls a JSP named show_email_entry.jsp that's stored in the same directory as the HTML page, and the Method attribute specifies that the HTTP Get method should be used with this action. Then, when the user clicks on the Submit button, the browser will send a request for the JSP.

You should also notice the Name attributes of the three text boxes that are used in the table within this HTML page. These are the names of the *parameters* that are passed to the JSP when the user clicks on the Submit button. In figure 4-1, the parameter names are firstName, lastName, and emailAddress and the parameter values are John, Smith, and jsmith@hotmail.com.

The code for the HTML page

```
<!doctype html public "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>

<head>
  <title>Chapter 4 - Email List application</title>
</head>

<body>
  <h1>Join our email list</h1>
  <p>To join our email list, enter your name and
     email address below. <br>
     Then, click on the Submit button.</p>

  <form action="show_email_entry.jsp" method="get">
  <table cellspacing="5" border="0">
    <tr>
      <td align="right">First name:</td>
      <td><input type="text" name="firstName"></td>
    </tr>
    <tr>
      <td align="right">Last name:</td>
      <td><input type="text" name="lastName"></td>
    </tr>
    <tr>
      <td align="right">Email address:</td>
      <td><input type="text" name="emailAddress"></td>
    </tr>
    <tr>
      <td></td>
      <td><br><input type="submit" value="Submit"></td>
    </tr>
  </table>
  </form>
</body>

</html>
```

Description

- The Action and Method attributes for the Form tag set up a request for a JSP that will be executed when the user clicks on the Submit button.

- The three text boxes represent *parameters* that will be passed to the JSP when the user clicks the Submit button.

- The parameter names are firstName, lastName, and emailAddress, and the parameter values are the strings that the user enters into the text boxes.

Figure 4-2   The code for the HTML page that calls the JSP

The code for the JSP

Figure 4-3 presents the code for the JSP. As you can see, much of the JSP code is HTML. In addition, though, Java code is embedded within the HTML code in the form of JSP *scriptlets* and *expressions*. Typically, a scriptlet is used to execute one or more Java statements while a JSP expression is used to display text. To identify scriptlets and expressions, JSPs use tags. To distinguish them from HTML tags, you can refer to them as *JSP tags*.

When you code a JSP, you can use the methods of the *request object* in your scriptlets or expressions. Since you don't have to explicitly create this object when you code JSPs, this object is sometimes referred to as the *implicit request object*. The scriptlet in this figure contains three statements that use the getParameter method of the request object. Each of these statements returns the value of the parameter that is passed to the JSP from the HTML page. Here, the argument for each getParameter method is the name of the textbox on the HTML page.

Once the scriptlet is executed, the values for the three parameters are available as variables to the rest of the page. Then, the three expressions can display these variables. Since these expressions are coded within the HTML tags for a table, the browser will display these expressions within a table.

After the table, the JSP contains some HTML that defines a form. This form contains only one control, a submit button named Return. When it is clicked, it takes the user back to the first page of the application. If you have any trouble visualizing how this button or the rest of the page will look when displayed by a browser, please refer back to figure 4-1.

As you read this book, remember that it assumes that you already know the basics of Java programming. If you have any trouble understanding the Java code in this chapter, you may need a refresher course on Java coding. To quickly review the basics of Java coding, we recommend that you use *Murach's Beginning Java 2* because it contains all the Java skills that you'll need for working with this book.

The code for the JSP

```html
<!doctype html public "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
  <title>Chapter 4 - Email List application</title>
</head>
<body>

<%
  String firstName = request.getParameter("firstName");
  String lastName = request.getParameter("lastName");
  String emailAddress = request.getParameter("emailAddress");
%>

<h1>Thanks for joining our email list</h1>

<p>Here is the information that you entered:</p>

  <table cellspacing="5" cellpadding="5" border="1">
    <tr>
      <td align="right">First name:</td>
      <td><%= firstName %></td>
    </tr>
    <tr>
      <td align="right">Last name:</td>
      <td><%= lastName %></td>
    </tr>
    <tr>
      <td align="right">Email address:</td>
      <td><%= emailAddress %></td>
    </tr>
  </table>

<p>To enter another email address, click on the Back <br>
button in your browser or the Return button shown <br>
below.</p>

<form action="join_email_list.html" method="post">
  <input type="submit" value="Return">
</form>

</body>
</html>
```

— **JSP scriptlet**

— **JSP expression**

Description

- Although a JSP looks much like an HTML page, a JSP contains embedded Java code.

- To code a *scriptlet* that contains one or more Java statements, you use the <% and %> tags. To display any *expression* that can be converted to a string, you use the <%= and %> tags.

- When you code a JSP, you can use the *implicit request object*. This object is named request. You can use the getParameter method of the request object to get the values of the parameters that are passed to the JSP.

Figure 4-3   The code for the JSP

How to create a JSP

Now that you have a general idea of how JSPs are coded, you're ready to learn some specific skills for creating a JSP. To start, you need to know more about coding scriptlets and expressions.

How to code scriptlets and expressions

Figure 4-4 summarizes the information you need for coding scriptlets and expressions within a JSP. To code a scriptlet, for example, you code Java statements that end with semicolons within the JSP scriptlet tags. To code an expression, you code any Java expression that evaluates to a string. Since primitive data types like integers or doubles are automatically converted to strings, you can also use expressions that evaluate to these data types.

When you're coding a scriptlet or an expression, you can use any of the methods of the implicit request object. In this figure, only the getParameter method is used, but you'll learn about two more methods of the request object in the next figure.

In this figure, the first two examples show different ways that you can display the value of a parameter. The first example uses a scriptlet to return the value of the firstName parameter and store it in a String object. Then, this example uses an expression to display the value. In contrast, the second example uses an expression to display the value of the firstName parameter without creating the firstName object.

The last example in this figure shows how two scriptlets and an expression can be used to display an HTML line five times while a Java variable within the HTML line counts from 1 to 5. Here, the first JSP scriptlet contains the code that begins a while loop. Then, a line of HTML code uses a JSP expression to display the current value of the counter for the loop. And finally, the second scriptlet contains the code that ends the loop.

The syntax for a JSP scriptlet

```
<% Java statements %>
```

The syntax for a JSP expression

```
<%= any Java expression that can be converted to a string %>
```

The syntax for getting a parameter from the implicit request object

```
request.getParameter(parameterName);
```

Examples that use scriptlets and expressions

A scriptlet and expression that display the value of the firstName parameter

```
<%
    String firstName = request.getParameter("firstName");
%>
The first name is <%= firstName %>.
```

An expression that displays the value of the firstName parameter

```
The first name is <%= request.getParameter("firstName") %>.
```

Two scriptlets and an expression that display an HTML line 5 times

```
<%
    int numOfTimes = 1;
    while (numOfTimes <= 5){
%>
    <h1> This line is shown <%= numOfTimes %> of 5 times in a JSP.</h1>
<%
        numOfTimes++;
    }
%>
```

Description

- Within a scriptlet, you can code one or more complete Java statements. Because these statements are Java statements, you must end each one with a semicolon.

- Within a JSP expression, you can code any Java expression that evaluates to a string. This includes Java expressions that evaluate to any of the primitive types, and it includes any object that has a toString method. Because a JSP expression is an expression, not a statement, you don't end it with a semicolon.

Figure 4-4   How to code scriptlets and expressions

How to use the methods of the request object

In the last figure, you learned how to use the getParameter method to return the value that the user entered into a textbox. Now, figure 4-5 summarizes that method and illustrates it in a new context. This figure also summarizes and illustrates two more methods of the implicit request object.

In most cases, the getParameter method returns the value of the parameter. For a textbox, that's usually the value entered by the user. But for a group of radio buttons or a combo box, that's the value of the button or item selected by the user.

For checkboxes or independent radio buttons that have a Value attribute, the getParameter method returns that value if the checkbox or button is selected and a null value if it isn't. For checkboxes or independent radio buttons that don't have a Value attribute, though, the getParameter method returns an "on" value if the checkbox or button is selected and a null value if it isn't. This is illustrated by the first example in this figure.

To retrieve multiple values for one parameter name, you can use the getParameterValues method as illustrated by the second example. This method is useful for controls like list boxes that allow multiple selections. After you use the getParameterValues method to return an array of String objects, you can use a loop to get the values from the array.

To get the names of all the parameters sent with a request, you can use the getParameterNames method to return an Enumeration object that contains the names. Then, you can search through the Enumeration object to get the parameter names, and you can use the getParameter method to return the value for each parameter name. This is illustrated by the third example.

If you're not familiar with the Enumeration class, you can learn more about it though the API. For most purposes, though, you only need to know that an Enumeration object is a collection that can be searched element by element. To determine if more elements exist in the collection, you can use the hasMoreElements method, which returns a boolean value. And to get the next element in the collection, you can use the nextElement method.

Three methods of the request object

Method	Description
getParameter(String param**)**	Returns the value of the specified parameter as a string if it exists or null if it doesn't. Often, this is the value defined in the Value attribute of the control in the HTML page or JSP.
getParameterValues(String param**)**	Returns an array of String objects containing all of the values that the given request parameter has or null if the parameter doesn't have any values.
getParameterNames()	Returns an Enumeration object that contains the names of all the parameters contained in the request. If the request has no parameters, the method returns an empty Enumeration object.

A scriptlet that determines if a checkbox is checked

```
<%
    String rockCheckBox = request.getParameter("Rock");
    // returns the value or "on" if checked, null otherwise.
    if (rockCheckBox != null){
%>
        You checked Rock music!
<% }
%>
```

A scriptlet that reads and displays multiple values from a list box

```
<%
    String[] selectedCountries = request.getParameterValues("country");
    // returns the values of items selected in list box.
    for (int i = 0; i < selectedCountries.length; i++){
%>
        <%= selectedCountries[i] %> <br>
<%
    }
%>
```

A scriptlet that reads and displays all request parameters and values

```
<%
    Enumeration parameterNames = request.getParameterNames();
    while (parameterNames.hasMoreElements()){
        String parameterName = (String) parameterNames.nextElement();
        String parameterValue = request.getParameter(parameterName);
%>
        <%= parameterName %> has value <%= parameterValue %>. <br>
<%
    }
%>
```

Description

- You can use the getParameter method to return the value of the selected radio button in a group or the selected item in a combo box. You can also use it to return the value of a selected check box or independent radio button, but that value is null if it isn't selected.

- If an independent radio button or a checkbox doesn't have a Value attribute, this method returns "on" if the control is selected or null if it isn't.

Figure 4-5   How to use the methods of the request object

Where and how to save a JSP

As figure 4-6 shows, you normally save the JSPs of an application in the same directory that you use for the HTML pages of the application. The difference is that the name for a JSP requires a jsp extension. So if you're using a text editor that's not designed for working with JSPs, you may have to place the filename in quotation marks to make sure the file is saved with the jsp extension.

Like HTML pages, JSPs must be saved in a directory that's available to the web server. For Tomcat 4.0, you can use any directory under the webapps directory. The root directory for the web applications that are presented in the first 16 chapters of this book is the webapps\murach directory, and the subdirectory for the Email List application that's presented in this chapter is email4. Note, however, that you can also use the webapps\ROOT directory that Tomcat sets up as the default root directory. Or, you can create your own subdirectories under the webapps directory.
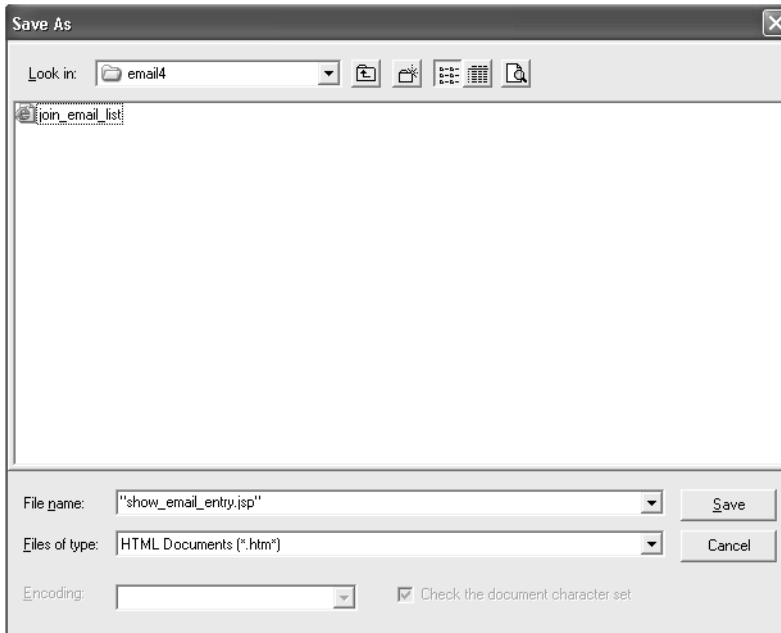
Where the show_email_entry.jsp page is saved

```
c:\tomcat\webapps\murach\email4
```

Other places you can save your JSPs

```
c:\tomcat\webapps\yourDocumentRoot
c:\tomcat\webapps\yourDocumentRoot\yourSubdirectory
c:\tomcat\webapps\ROOT
c:\tomcat\webapps\ROOT\yourSubdirectory
```

A standard dialog box for saving a JSP

Description

- JSPs are normally saved in the same directory as the HTML pages. This directory should be a subdirectory of the web applications directory for your server. If you're running Tomcat on your PC, that directory is usually c:\tomcat\webapps or c:\jakarta-tomcat\webapps.

- For the first 16 chapters of this book, the document root directory for all applications is the murach directory. As a result, the HTML and JSP files for each application are stored in this directory or one of its subdirectories.

- If you're using Tomcat on your local system, you can also use webapps\ROOT as the root directory for your applications. The ROOT directory is automatically set up when you install Tomcat, and it is the default document root directory.

- To make sure that the filename for a JSP is saved with the jsp extension when you're using an HTML or text editor, you can enter the filename within quotes.

Figure 4-6   Where and how to save a JSP

How to request a JSP

After you create a JSP, you need to test it. One way to do that is to click on a link or a button on an HTML page that requests the JSP. Another way is to enter a URL into a web browser that requests the JSP. Figure 4-7 shows how to request a JSP either way.

To request a JSP from an HTML form, you use the Action attribute of the form to provide a path and filename that point to the JSP. This is illustrated by the first example in this figure. Here, the assumption is that the HTML page and the JSP are in the same directory. If they weren't, you would have to supply a relative or absolute path for the JSP file.

To request a JSP by entering its URL into a browser, you enter an absolute URL as shown by the next two examples in this figure. The first example shows the URL for the JSP when it's stored on a local web server in the email4 directory of the murach directory. The second example shows the URL for the JSP if the JSP was deployed on the Internet server for *www.murach.com*.

When you test a JSP by entering a URL, you will often want to pass parameters to it. To do that, you can add the parameters to the end of the URL as shown by the last examples in this figure. Here, the question mark after the jsp extension indicates that one or more parameters will follow. Then, you code the parameter name, the equals sign, and the parameter value for each parameter that is passed, and you separate multiple parameters with ampersands (&). If you omit a parameter that's required by the JSP, the getParameter method will return a null value for that parameter.

When you use a Get method to request a JSP from another page, any parameters that are passed to the JSP will be displayed in the browser's URL address. In this figure, for example, you can see the first two parameters that have been attached to the URL. However, in the next figure, you'll learn that the Post method works differently.

A URL that includes parameters
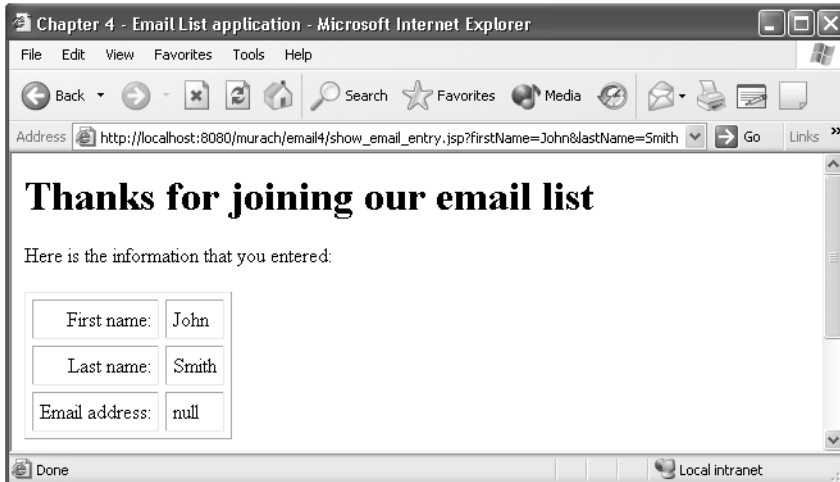
How Tomcat maps directories to HTTP calls

Tomcat directory	URL
c:\tomcat\webapps\murach	http://localhost:8080/murach/
c:\tomcat\webapps\murach\email4	http://localhost:8080/murach/email4
c:\tomcat\webapps\ROOT	http://localhost:8080/
c:\tomcat\webapps\ROOT\email4	http://localhost:8080/email4

A Form tag that requests a JSP

```
<form action="show_email_entry.jsp" method="get">
```

Two URLs that request a JSP

```
http://localhost:8080/murach/email4/show_email_entry.jsp
http://www.murach.com/email4/show_email_entry.jsp
```

How to include parameters

```
show_email_entry.jsp?firstName=John
show_email_entry.jsp?firstName=John&lastName=Smith
```

Description

- When you use the Get method to request a JSP from an HTML form, the parameters are automatically appended to the URL.
- When you code or enter a URL that requests a JSP, you can add a parameter list to it starting with a question mark and with no intervening spaces. Then, each parameter consists of its name, an equals sign, and its value. To code multiple parameters, use ampersands (&) to separate the parameters.

Figure 4-7   How to request a JSP

When to use the Get and Post methods

When you code a Form tag that requests a JSP, you can code a Method attribute that specifies the HTTP method that's used for the request. The Get method is the default HTTP method, but the Post method is also commonly used.

Figure 4-8 presents the pros and cons of using the Get and Post methods. With either method, you can still test the page by appending the parameters to the URL string. So the question really comes down to selecting the appropriate method for the finished web application.

There are two primary reasons for using the Post method. First, since the Post method doesn't append parameters to the end of the URL, it is more appropriate for working with sensitive data. If, for example, you're passing a parameter for a password or a credit card number, the Post method prevents these parameters from being displayed in the browser. In addition, it prevents the web browser from including these parameters in a bookmark for a page. Second, you need to use the Post method if your parameters contain more than 4 KB of data.

For all other uses, the Get method is preferred. It runs slightly faster than the Post method, and it lets the user bookmark the page along with the parameters that were sent to the page.

An HTML form tag that uses the Post method

```
<form action="show_email_entry.jsp" method="post">
```

A JSP that's requested through the Post method
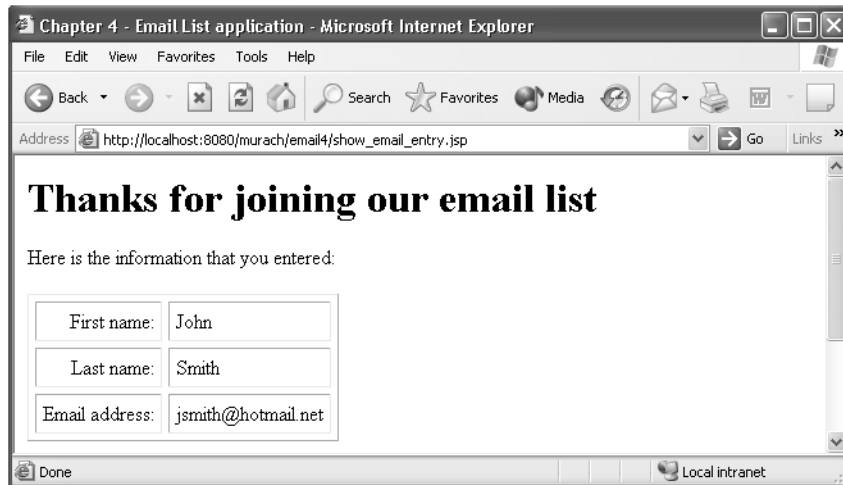
When to use the Get method

- If you want to transfer data as fast as possible.
- If the HTML form only needs to transfer 4 KB of data or less.
- If it's okay for the parameters to be displayed in the URL.
- If you want users to be able to include parameters when they bookmark a page.

When to use the Post method

- If you're transferring over 4 KB of data.
- If it's not okay for the parameters to be appended to the URL.

Description

- The visible difference between the Get and Post methods is the URL that's displayed in the browser. For Get requests from an HTML page, the parameters are appended to the URL. For Post requests, the parameters are still sent, but they're not displayed in the browser.
- You can test a JSP that uses either method by appending the parameters to the URL.

Figure 4-8   When to use the Get and Post methods

How to use regular Java classes with JSPs

In this topic, you'll learn how to use regular Java classes to do the processing that a JSP requires. In particular, you'll learn how to use two classes named User and UserIO to do the processing for the JSP of the Email List application.

The code for the User and UserIO classes

Figure 4-9 presents the code for a business class named User and an I/O class named UserIO. The package statement at the start of each class indicates where each class is stored. Here, the User class is stored in the business directory because it defines a business object while the UserIO class is stored in the data directory because it provides the data access for the application.

The User class defines a user of the application. This class contains three instance variables: firstName, lastName, and emailAddress. It includes a constructor that accepts three values for these instance variables. And it includes get and set methods for each instance variable.

In contrast, the UserIO class contains one static method named addRecord that writes the values stored in a User object to a text file. This method accepts two parameters: a User object and a string that provides the path for the file. If this file exists, the method will add the user data to the end of it. If the file doesn't exist, the method will create it and add the data at the beginning of the file.

If you've read the first six chapters of *Murach's Beginning Java 2*, you should understand the code for the User class. And if you've read chapters 16 and 17, you should understand the code in the UserIO class. The one exception is the use of the synchronized keyword in the addRecord method declaration. But this keyword just prevents two users from writing to the file at the same time, which could lead to an error.

The code for the User class

```java
package business;

public class User{
    private String firstName;
    private String lastName;
    private String emailAddress;

    public User(){}

    public User(String first, String last, String email){
        firstName = first;
        lastName = last;
        emailAddress = email;
    }

    public void setFirstName(String f){
        firstName = f;
    }
    public String getFirstName(){ return firstName; }

    public void setLastName(String l){
        lastName = l;
    }
    public String getLastName(){ return lastName; }

    public void setEmailAddress(String e){
        emailAddress = e;
    }
    public String getEmailAddress(){ return emailAddress; }
}
```

The code for the UserIO class

```java
package data;

import java.io.*;
import business.User;

public class UserIO{
    public synchronized static void addRecord(User user, String filename)
                                    throws IOException{
        PrintWriter out = new PrintWriter(
                        new FileWriter(filename, true));
        out.println(user.getEmailAddress()+ "|"
                    + user.getFirstName() + "|"
                    + user.getLastName());
        out.close();
    }
}
```

Note

- The synchronized keyword in the declaration for the addRecord method of the UserIO class prevents two users of the JSP from using that method at the same time.

Figure 4-9   The code for the User and UserIO classes

Where and how to save and compile regular Java classes

If you're using Tomcat 4.0, figure 4-10 shows where and how to save your compiled Java classes (the .class files) so Tomcat can access them. Usually, you'll save your source code (the .java files) in the same directory, but that's not required.

The two paths shown at the top of this figure show where the User and UserIO classes that come with this book are saved. After that, the figure presents the syntax for other paths that can be used to store Java classes. If you review these paths, you'll see that each one places the Java classes in a subdirectory of the WEB-INF\classes directory.

Since the User class contains a package statement that corresponds to the business directory, it must be located in the WEB-INF\classes\business directory. In contrast, if the package statement specified "murach.email", the compiled classes would have to be located in the WEB-INF\classes\murach\email directory.

Since TextPad is designed for working with Java, you can use it to compile regular Java classes. However, you may need to configure your system as described in appendix A before it will work properly. In particular, you may need to add the appropriate WEB-INF\classes directory to your classpath.

If you use the DOS prompt window to compile your classes, you can do that as shown in this figure. Here, a DOS prompt is used to compile the User class. To start, the cd command changes the current directory to the \WEB-INF\classes directory. Then, the javac command is used with "business\User.java" as the filename. This compiles the User class and stores it in the business package, which is what you want.

Where the User class is saved

```
c:\tomcat\webapps\murach\WEB-INF\classes\business
```
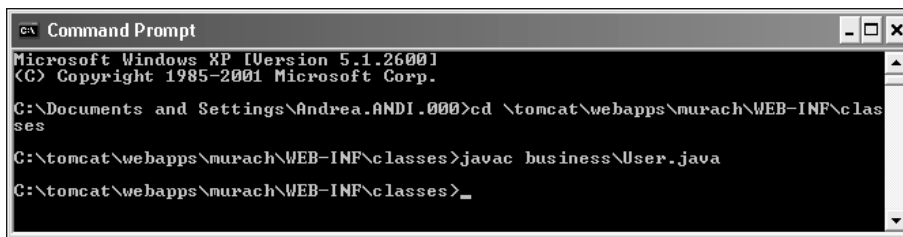
Where the UserIO class is saved

```
c:\tomcat\webapps\murach\WEB-INF\classes\data
```

Other places to save your Java classes

```
c:\tomcat\webapps\yourDocumentRoot\WEB-INF\classes
c:\tomcat\webapps\yourDocumentRoot\WEB-INF\classes\packageName
c:\tomcat\webapps\ROOT\WEB-INF\classes
c:\tomcat\webapps\ROOT\WEB-INF\classes\packageName
```

The DOS prompt window for compiling the User class

```
Command Prompt                                              _ □ x
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Andrea.ANDI.000>cd \tomcat\webapps\murach\WEB-INF\clas
ses

C:\tomcat\webapps\murach\WEB-INF\classes>javac business\User.java

C:\tomcat\webapps\murach\WEB-INF\classes>_
```

Description

- Although you can save the source code (the .java files) in any directory, you must save the class files (the .class files) in the WEB-INF\classes directory or one of its subdirectories. This can be subordinate to the ROOT directory or your own document root directory.

- To compile a class, you can use TextPad's Compile Java command, your IDE's compile command, or the javac command from the DOS prompt window.

- If you have trouble compiling a class, make sure your system is configured correctly as described in appendix A.

Figure 4-10   Where and how to save and compile regular Java classes

A JSP that uses the User and UserIO classes

Figure 4-11 shows the code for the JSP in the Email List application after it has been enhanced so it uses the User and UserIO classes to process the parameters that have been passed to it. In the first statement of the body, a special type of JSP tag is used to import the business and data packages that contain the User and UserIO classes. You'll learn how to code this type of tag in the next figure. For now, though, you should focus on the other shaded lines in this JSP.

In the scriptlet of the JSP, the getParameter method is used to get the values of the three parameters that are passed to it, and these values are stored in String objects. Then, the next statement uses these strings as arguments for the constructor of the User class. This creates a User object that contains the three values. Last, this scriptlet uses the addRecord method of the UserIO class to add the three values of the User object to a file named UserEmail.txt that's stored in the WEB-INF\etc directory. Since the WEB-INF\etc directory isn't web-accessible, this prevents users of the application from accessing this file.

After the scriptlet, the code in the JSP defines the layout of the page. Within the HTML table definitions, the JSP expressions use the get methods of the User object to display the first name, last name, and email address values. Although these JSP expressions could use the String objects instead, the code in this figure is intended to show how the get methods can be used.

The code for a JSP that uses the User and UserIO classes

```
<!doctype html public "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>

<head>
  <title>Chapter 4 - Email List application</title>
</head>

<body>
<%@ page import="business.*, data.*" %>
<%
    String firstName = request.getParameter("firstName");
    String lastName = request.getParameter("lastName");
    String emailAddress = request.getParameter("emailAddress");

    User user = new User(firstName, lastName, emailAddress);
    UserIO.addRecord(user, "../webapps/murach/WEB-INF/etc/UserEmail.txt");
%>

<h1>Thanks for joining our email list</h1>

<p>Here is the information that you entered:</p>

  <table cellspacing="5" cellpadding="5" border="1">
    <tr>
      <td align="right">First name:</td>
      <td><%= user.getFirstName() %></td>
    </tr>
    <tr>
      <td align="right">Last name:</td>
      <td><%= user.getLastName() %></td>
    </tr>
    <tr>
      <td align="right">Email address:</td>
      <td><%= user.getEmailAddress() %></td>
    </tr>
  </table>

<p>To enter another email address, click on the Back <br>
button in your browser or the Return button shown <br>
below.</p>

<form action="join_email_list.html" method="post">
  <input type="submit" value="Return">
</form>

</body>
</html>
```

Description

- This JSP uses a scriptlet to create a User object and add it to a file, and it uses JSP expressions to display the values of the User object's instance variables.
- Since the User and UserIO classes are stored in the business and data packages, the JSP must import these packages as shown in figure 4-12.

Figure 4-11   A JSP that uses the User and UserIO classes

How to use three more types of JSP tags

So far, you've learned how to use the JSP tags for scriptlets and expressions. Now, you'll learn how to use three more types of JSP tags. All five types of JSP tags are summarized at the top of figure 4-12.

How to import classes

Figure 4-12 also shows how to use a *JSP directive* to import classes in a JSP. The type of directive that you use for doing that is called a *page directive*, and the shaded statement at the start of the JSP shows how to code one.

To code a page directive for importing classes, you code the starting tag and the word *page* followed by the Import attribute. Within the quotation marks after the equals sign for this attribute, you code the names of the Java classes that you want to import just as you do in a Java import statement. In the example in this figure, all the classes of the business and data packages are imported, plus the Date class in the java.util package.

Once the page directive imports the packages, the JSP can access the User and UserIO classes and the Date class. The scriptlet that follows creates a User object from the User class and uses the addRecord method of the UserIO class to write the data for the User object to a text file. The last line in this example uses the default constructor of the Date class as an expression in an HTML line. This works because the JSP will automatically convert the Date object that's created by the constructor into a string.

The JSP tags presented in this chapter

Tag	Name	Purpose
`<% %>`	JSP scriptlet	To insert a block of Java statements.
`<%= %>`	JSP expression	To display the string value of an expression.
`<%@ %>`	JSP directive	To set conditions that apply to the entire JSP.
`<%-- --%>`	JSP comment	To tell the JSP engine to ignore code.
`<%! %>`	JSP declaration	To declare instance variables and methods for a JSP.

JSP code that imports Java classes

```
<%@ page import="business.*, data.*, java.util.Date" %>

<%
    String firstName = request.getParameter("firstName");
    String lastName = request.getParameter("lastName");
    String emailAddress = request.getParameter("emailAddress");

    User user = new User(firstName, lastName, emailAddress);
    UserIO.addRecord(user, "../webapps/murach/WEB-INF/etc/UserEmail.txt");
%>

Today's date is <%= new Date() %>.
```

Description

- To define the conditions that the JSP engine should follow when converting a JSP into a servlet, you can use a *JSP directive*.

- To import classes in a JSP, you use the import attribute of the *page directive*. This makes the imported classes available to the entire page.

- You can also use the page directive to define other conditions like error handling and content type conditions. You'll learn more about this directive throughout this book.

Figure 4-12   How to import classes

How to code comments in a JSP

Figure 4-13 shows how to code *JSP comments*, and it shows how you can code Java comments and HTML comments in a JSP. As a result, you need to understand how the three types of comments are processed.

Like Java comments, any code within a JSP comment isn't compiled or executed. In contrast, any code within an HTML comment is compiled and executed, but the browser doesn't display it. This is illustrated by the HTML comment in this figure. Here, the HTML line within the comment isn't displayed by the browser, but the integer variable in the comment is incremented each time the page is accessed. To prevent this integer variable from being incremented, you have to use a JSP comment instead of an HTML comment.

A JSP comment

```
<%--
   Today's date is <%= new java.util.Date() %>.
--%>
```

Java comments in a JSP scriptlet

```
<%
   //These statements retrieve the request parameter values
   String firstName = request.getParameter("firstName");
   String lastName = request.getParameter("lastName");
   String emailAddress = request.getParameter("emailAddress");

   /*
   User user = new User(firstName, lastName, emailAddress);
   UserIO.addRecord(user, "../webapps/murach/WEB-INF/etc/UserEmail.txt");
   */
%>
```

An HTML comment in a JSP

```
<!--
   <i>This page has been accessed <%= ++i %> times.</i>
-->
The value of the variable i is <%= i %>.
```

Description

- When you code *JSP comments*, the comments aren't compiled or executed.
- When you code Java comments within a scriptlet, the comments aren't compiled or executed.
- When you code HTML comments, the comments are compiled and executed, but the browser doesn't display them.

Figure 4-13   How to code comments in a JSP

How to declare instance variables and methods

When a JSP is requested for the first time, one *instance* of the JSP is created and loaded into memory, and a *thread* is started that executes the Java code in the JSP. For each subsequent request for the JSP, another thread is started that can access the one instance of the JSP. When you code variables in scriptlets, each thread gets its own copy of each variable, which is usually what you want.

In some cases, though, you may want to declare instance variables and methods that can be shared between all of the threads that are accessing a JSP. To do that, you can code *JSP declarations* as shown in figure 4-14. Then, the instance variables and methods are initialized when the JSP is first requested. After that, each thread can access those instance variables and methods.

This is illustrated by the accessCount variable that's declared as an instance variable in this figure. This variable is incremented by one each time the JSP is requested. Once it's incremented, the current value is stored in a local variable. Later, when the variable is displayed, it represents the total number of times that the page has been accessed.

In this figure, the synchronized keyword prevents two threads from using the method or modifying the instance variable at the same time. This results in a *thread-safe* JSP. Without the synchronized keyword for the method, for example, two or more threads could write to the text file at the same time, which could lead to an I/O error if one thread tries to open the file before the previous thread closes it. Without the synchronized and this keywords for the block of code, two or more threads could update the accessCount variable at the same time. If, for example, user 4 updates the accessCount before it is stored in the localCount variable, the access count that's displayed for user 3 will be 4. Although that's highly unlikely and insignificant in this case, this illustrates the type of problem that can occur.

In most cases, though, it's okay for multiple threads to access a method or instance variable at the same time because no harm can come from it. That's why you only need to code a thread-safe JSP when multiple threads could cause inaccurate or faulty results. The potential for this often arises when an instance variable is modified by a thread or when a method accesses a data store. Then, you can decide whether you need to provide for this by coding a thread-safe JSP.

If you find your JSP becoming cluttered with declarations, you should consider restructuring your program. In some cases, you may want to use regular Java classes like the ones shown in this chapter. In other cases, you may want to use a servlet as described in the next chapter. Better yet, you may want to use a combination of servlets, JSPs, and other Java classes as described in chapter 6.

JSP code that declares an instance variable and a method

```
<%@ page import="business.*, data.*, java.util.Date, java.io.*" %>

<%! int accessCount = 0; %>
<%!
    public synchronized void addRecord(User user, String filename)
                            throws IOException{
        PrintWriter out = new PrintWriter(
                            new FileWriter(filename, true));
        out.println(user.getEmailAddress()+ "|"
                    + user.getFirstName() + "|"
                    + user.getLastName());
        out.close();
    }
%>
<%
    String firstName = request.getParameter("firstName");
    String lastName = request.getParameter("lastName");
    String emailAddress = request.getParameter("emailAddress");

    User user = new User(firstName, lastName, emailAddress);
    addRecord(user, "../webapps/murach/WEB-INF/etc/UserEmail.txt");

    int localCount = 0;
    synchronized (this) {
        accessCount++;
        localCount = accessCount;
    }
%>
.
.
.
<p><i>This page has been accessed <%= localCount %> times.</i></p>
```

Description

- You can use *JSP declarations* to declare instance variables and methods for a JSP. Unlike the variables defined in scriptlets, one set of instance variables and methods are used by all the users of the JSP.

- To code a *thread-safe* JSP, you must synchronize access to instance variables and methods that should only be called by one thread at a time. Otherwise, two threads may conflict when they try to modify the same instance variable at the same time or when they try to execute the same method that accesses a data store at the same time.

- To synchronize access to a method, you can use the synchronized keyword in the method declaration. To synchronize access to a block of code, you can use the synchronized keyword and the this keyword before the block.

Figure 4-14   How to declare instance variables and methods

# A JSP that imports classes and declares instance variables

To illustrate how these JSP tags work together, figure 4-15 presents an enhanced version of the Email List application. Here, the page that's displayed is the same as the page shown in figure 4-1 except that it displays two lines below the Return button. The first line displays the date and time that the user accessed the page. The second displays the number of times the page has been accessed since it was initialized.

In the code for the JSP, the page directive imports the required classes including the User and UserIO classes from the business and data packages. This directive is followed by two declarations. The first declares an instance variable for the number of users that access the JSP. The second declares an instance variable for the path of the text file that will store the user data. The scriptlet that follows is one that you've already seen so you should understand how it works.

Since you've already seen the HTML tags that display the data for this JSP, this figure doesn't show these tags. They are the same HTML tags that are shown in figure 4-11.

The first shaded line at the end of this JSP uses the constructor for the Date class to return the current date and time. Since this constructor is coded within an expression, each user of the JSP will get its own copy of the date and time. As a result, the HTML line will display a different time for each user. In contrast, as explained earlier, the second shaded line displays the value of an instance variable that is incremented by one for each thread.

A JSP that displays the date and an instance variable
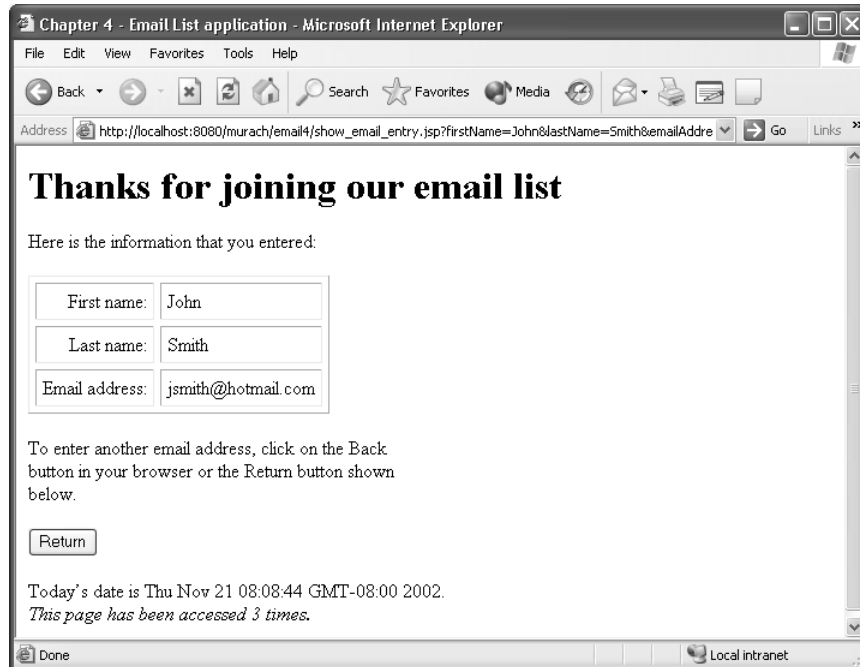
The code for the JSP

```html
<!doctype html public "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
  <title>Chapter 4 - Email List application</title>
</head>

<body>
<%@ page import= "business.User, data.UserIO, java.util.Date, java.io.*" %>
<%! int accessCount = 0; %>
<%! String file = "../webapps/murach/WEB-INF/etc/UserEmail.txt"; %>
<%   String firstName = request.getParameter("firstName");
     String lastName = request.getParameter("lastName");
     String emailAddress = request.getParameter("emailAddress");
     User user = new User(firstName, lastName, emailAddress);
     UserIO.addRecord(user, file);
     int localCount = 0;
     synchronized (this) {
         accessCount++;
         localCount = accessCount;
     }
%>

<!-- missing code that's the same as figure 4-11 -->

Today's date is <%= new Date() %>. <br>
<i>This page has been accessed <%= localCount %> times.</i>

</body>
</html>
```

Figure 4-15   A JSP that imports classes and declares instance variables

How to work with JSP errors

When you develop JSPs, you will inevitably encounter errors. That's why the last three figures in this chapter show you how to work with JSP errors.

How to debug JSP errors

Figure 4-16 presents the two most common JSP errors. HTTP Status 404 means that Tomcat received the HTTP request but couldn't find the requested resource. You've already seen this in chapter 2. To fix this type of problem, make sure that you've entered the correct path and filename for the request and that the requested file is in the right location.

In contrast, HTTP Status 500 means that the server received the request and found the resource but couldn't fill the request. This usually means that the JSP engine wasn't able to compile the JSP due to a coding error in the JSP. To fix this type of problem, you can review the other information provided by the error page. In the error page in this figure, for example, you can see that a semicolon is missing at the end of one of the statements in the JSP scriptlet.

To correct this type of error, you should fix the JSP so it will compile correctly. To do this, you can open the JSP, add the semicolon, save it, and refresh the screen. Then, the JSP engine will recognize that the JSP has been modified and it will automatically attempt to compile and load the JSP.

This figure also gives some tips for debugging JSP errors. The first three tips will help you solve most status 404 errors. The fourth one will help you find the cause of most status 500 errors. And when all else fails, you can look at the servlet that's generated from the JSP as explained in figure 4-18.

An error page for a common JSP error
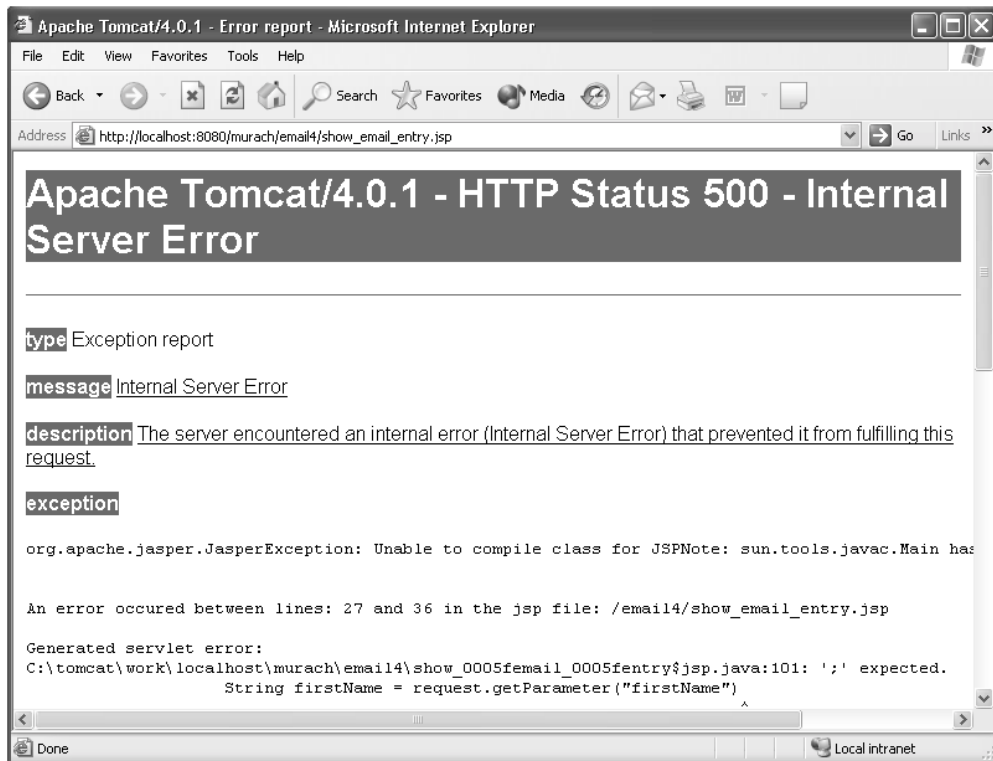
Common JSP errors

- HTTP Status 404 – File Not Found Error
- HTTP Status 500 – Internal Server Error

Tips for debugging JSP errors

- Make sure the Tomcat server is running.
- Make sure that the URL is valid and that it points to the right location for the requested page.
- Make sure all of the HTML, JSP, and Java class files are in the correct locations.
- Read the error page carefully to get all available information about the error.
- As a last resort, look at the servlet that's generated for the JSP as shown in figure 4-18.

Figure 4-16   How to debug JSP errors

How to use a custom error page

As you have just seen, the server displays its *error page* when an uncaught exception is thrown at run time. Although that's okay while you're developing a web application, you may not want error pages like that displayed once the application is put into production. In that case, you can design your own error pages and use them as shown in figure 4-17. That makes the user interface easier to understand.

To designate the error page that should be used when any uncaught exception is thrown by a JSP page, you use the errorPage attribute of a page directive. This attribute designates the JSP or HTML page that you want to use as the error page. This is illustrated by the first example in this figure.

Then, if you want the error page to have access to the *implicit exception object*, you set the isErrorPage attribute of its page directive to true as illustrated by the second example in this figure. This makes an object named exception available to the page. In the second example, you can see how this object is used in a JSP expression to display the exception type and description. (When you code an object name in an expression, its toString method is invoked.)

A custom error page in a browser
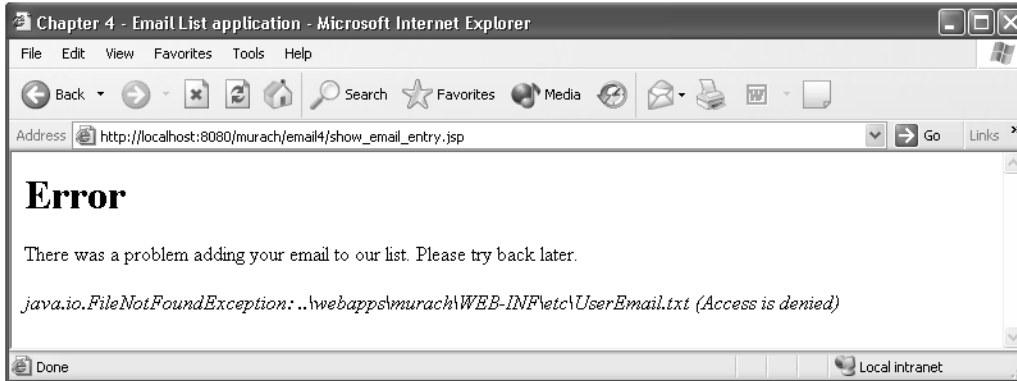
Two more attributes of a page directive

Attribute	Use
errorPage	To designate an error page for any uncaught exceptions that are thrown by the page.
isErrorPage	To identify a page as an error page so it has access to the implicit exception object.

A page directive that designates a custom error page

```
<%@ page errorPage="show_error_message.jsp" %>
```

A JSP that is designated as a custom error page

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
<head>
    <title>Chapter 4 - Email List application</title>
</head>

<body>
    <%@ page isErrorPage="true" %>
    <h1>Error</h1>
    <p>There was a problem adding your email to our list.
       Please try back later.</p>
    <p><i><%= exception %></i></p>
</body>
</html>
```

Description

- You can use the errorPage attribute of a JSP page directive to designate an error page that is requested if an uncaught exception is thrown at runtime. That error page can be either a JSP or an HTML page.

- If you want the error page to have access to the JSP *implicit exception object*, the error page must be a JSP that includes a page directive with the isErrorPage attribute set to true. Since the implicit exception object is an object of the java.lang.Throwable class, you can call any methods in this class from the exception object.

Figure 4-17   How to use a custom error page

When and how to view the servlet that's generated for a JSP

As you work with JSPs, you should keep in mind that JSPs are translated into servlets and compiled before they are run. As a result, when a Status 500 error occurs, it's usually because the JSP engine can't compile the generated servlet, even though that problem is caused by bad code in the related JSP.

As you have seen in figure 4-16, you can usually find the cause of a Status 500 error by reading the error page and proceeding from there. As a last resort, though, it sometimes makes sense to view the servlet that has been generated from the JSP. For instance, figure 4-18 shows the servlet that's generated for the JSP in figure 4-15.

Before you take a closer look at that servlet, though, you need to understand how a servlet is generated, compiled, and instantiated. In particular, you need to keep these facts in mind. First, each JSP is translated into a servlet class. Second, the servlet class is compiled when the first user requests the JSP. Third, one instance of the servlet class is instantiated when the first user requests the JSP. Fourth, each user is treated as a thread by the servlet, and each thread gets its own copy of the local variables of the servlet methods. With this conceptual background, the generated servlet should make more sense.

At the start of this figure, you can see the path and filename for the servlet that's generated from the JSP in figure 4-15. Although all generated servlets are stored in the work\localhost directory for Tomcat 4.0, the rest of the path is dependent on the location of the actual JSP. If, for example, the JSP is saved in the email4 directory of the murach application directory, then the generated servlet is saved in the work\localhost\murach\email4 directory. The filename of this servlet is generated from the JSP filename.

If you find the servlet file and open it with a text editor, you will probably find the code overwhelming at first. But the partial code in this figure should give you an idea of what's going on. Somewhere near the start of the code, you'll find the instance variables of the JSP declarations declared as variables of the class.

After that, you'll find a JSP service method that contains all of the code from the JSP scriptlet. Because this method is called for each user, each user (or thread) will get a copy of the local variables used by this method. Then, after the code that's generated for the scriptlets, you'll find out.write statements that return HTML to the browser and out.print statements that return Java expressions to the browser.

After you read the next chapter, which shows you how to develop servlets, the code for this servlet should make more sense. But if you compare the code for the JSP in figure 4-15 with its servlet code, you should get a good idea of what's going on. Remember, though, that you usually don't need to view the generated servlets for debugging purposes.

The location of the servlet class for the JSP in figure 4-15

```
C:\tomcat\work\localhost\murach\email4\show_0005femail_0005fentry$jsp.java
```

Part of the servlet class that's generated from the JSP in figure 4-15

```
public class show_0005femail_0005fentry$jsp extends HttpJspBase {

    int accessCount = 0;
    String file = "../webapps/murach/WEB-INF/etc/UserEmail.txt";
    ...
    }

    public void _jspService(HttpServletRequest request, HttpServletResponse
        response) throws java.io.IOException, ServletException {
        ...
        String firstName = request.getParameter("firstName");
        String lastName = request.getParameter("lastName");
        String emailAddress = request.getParameter("emailAddress");

        User user = new User(firstName, lastName, emailAddress);
        addRecord(user, file);
        int localCount = 0;
        synchronized (this) {
            accessCount++;
            localCount = accessCount;
        }
        ...
        out.write("... <td align=\"right\">First name:</td> ...");
        out.print( firstName );
        ...
    }
}
```

Instance variables from JSP declarations

Local data from a JSP scriptlet

A write statement for HTML tags

A print statement from a JSP expression

How a servlet is generated, compiled, and instantiated from a JSP

- When a JSP is first requested, the JSP engine translates the page into a servlet class and compiles the class. When it does, it places all JSP scriptlets into a service method. As a result, all variables defined by the scriptlets are local variables to the service method.

- After the servlet class is compiled, one instance of the class is instantiated and a new thread is created. Then, the service method is called.

- For each subsequent user that requests the JSP, a new thread is created from the instance of the servlet class and the service method is called. This method delivers a copy of the local variables for each thread, but all threads share the instance variables.

When and how to view the servlet that's generated for a JSP

- Although you need to know that your JSPs are translated into servlets before they are executed, you normally don't need to view them. For advanced debugging problems, though, it is sometimes useful to view the servlet for a JSP.

- If you're using Tomcat 4.0 or later, you can find the code for the Java servlet class in the subdirectory of the tomcat\work\localhost directory. Starting with that directory, you'll find a directory structure that corresponds to the one for your JSPs.

- If you're using another JSP engine, you should find the servlet class in a similar directory structure.

Figure 4-18   When and how to view the servlet that's generated for a JSP

Perspective

The goal of this chapter has been to provide you with the basics of coding, saving, and viewing JSPs. At this point, you should be able to code simple, but practical, JSPs of your own. In addition, you should understand how HTML pages communicate with JSPs, how JSPs communicate with regular Java classes, and how to fix some of the most common JSP errors.

In the next chapter, you'll learn how to use the same types of skills with servlets. In fact, the next chapter uses the same application that's presented in this chapter, but it uses a servlet instead of a JSP.

Summary

- A *JavaServer Page*, or *JSP*, consists of HTML code plus Java code that's embedded in *scriptlets* and *expressions*. Within the scriptlets and expressions, you can use the methods of the *implicit request object* to get the *parameters* that are passed to the JSP.

- The JSPs for an application are stored in the document root directory or one of its subdirectories. Then, you can view a JSP by entering a URL that corresponds to its directory location.

- When you use the Get method to pass parameters to a JSP, the parameters are displayed in the URL. When you use the Post method, they aren't. Although the Get method transfers data faster than the Post method, you can't use the Get method to transfer more than 4K of data.

- You can use business and data classes from within scriptlets and expressions just as you use them from Java classes. However, the business and data classes must be stored in the WEB-INF\classes directory or one of its subdirectories.

- You use a *JSP directive* known as a *page directive* to import classes for use in a JSP.

- When you use *JSP comments*, the comments aren't compiled or executed. In contrast, HTML comments are compiled and executed, but the browser doesn't display them.

- You use *JSP declarations* to declare instance variables and methods for a JSP. To code a *thread-safe* JSP, you must synchronize access to these instance variables and methods.

- You can use a page directive to designate an error page for an uncaught exception. You can code this directive so it has access to the JSP *implicit exception object* and you can use this object's methods.

- When a JSP is first requested, the JSP engine translates the page into a servlet and compiles it. Then, one instance of the class is instantiated. For each subsequent request, a new thread is created from this single instance of the servlet class. This means that each requestor gets its own copy of the local variables, but all requestors share the same instance variables.

Terms

JavaServer Page (JSP)	implicit request object	JSP declaration
parameter	JSP directive	thread-safe
scriptlet	page directive	error page
JSP expression	JSP comment	implicit exception
JSP tag	instance	object
request object	thread	

Objectives

- Code and test JavaServer Pages that require any of the features presented in this chapter including scriptlets, expressions, page directives, JSP comments, JSP declarations, and custom error pages.

- Describe the directory structure that should be used for JSPs, business, and data classes.

- List two differences between the Get and Post methods that can be used to pass parameters to a JSP.

- Describe the difference between JSP comments and HTML comments.

- Explain what is meant by a thread-safe JSP and describe what you have to do to develop one.

- Describe the process of running a JSP and its effect on local and instance variables.

Exercise 4-1   Create a custom error page

In this exercise, you'll create a custom error page for the Email List application.

1. Run the HTML document named join_email_list.html that's in the webapps\murach\email4 directory so it accesses and runs the JSP named show_email_entry.jsp.

2. Create a custom error page named show_error_page.jsp that displays the type of error, the current date, and the number of times the error page has been accessed. Save this JSP in the webapps\murach\email4 directory.

3. Edit the JSP file named show_email_entry.jsp that's in the email4 directory so it uses the show_error_page.jsp.

4. Test the custom error page. To do that, you can cause an exception by changing the properties of the file so it's read-only. In Windows, you can do this by right clicking the file icon in Explorer, selecting Properties, and selecting Read-only.

Exercise 4-2 Create a new JSP

In this exercise, you'll modify the HTML document for the Email List application, and you'll create a new JSP that responds to the HTML document.

1. Modify the HTML document named join_email_list.html that's in the email4 directory so it has this line of text after the Email address box: "I'm interested in these types of music." Then, follow this line with a list box that has options for Rock, Country, Bluegrass, and Folk music. This list box should be followed by the Submit button, and the Submit button should link to a new JSP named show_music_choices.jsp

2. Create a new JSP named show_music_choices.jsp that responds to the changed HTML document. This JSP should start with an H1 line that reads like this:

   Thanks for joining our email list, John Smith.

   And this line should be followed by text that looks like this:

   We'll use email to notify you whenever we have new releases for these types of music:

   Country
   Bluegrass

   In other words, you list the types of music that correspond to the items that are selected in the list box. And the entire web page consists of just the heading and text lines that I've just described.

3. Test the HTML document and the new JSP by running them. Note the parameter list that is passed to the JSP by the HTML document. Then, test the new JSP by using a URL that includes a parameter list.