TRAINING & REFERENCE

murach's Java servlets and JSP

(Chapter 5)

Andrea Steelman
Joel Murach

MIKE MURACH & ASSOCIATES, INC.
1-800-221-5528 • (559) 440-9071 • Fax: (559) 440-0963
murachbooks@murach.com • www.murach.com

Book contents

5

How to develop servlets

In chapter 4, you learned how to develop a web application that consisted of an HTML page and a JavaServer Page. In this chapter, you'll learn how to create the same application using a servlet instead of a JSP. When you complete this chapter, you should be able to use servlets to develop simple web applications of your own.

The Email List application

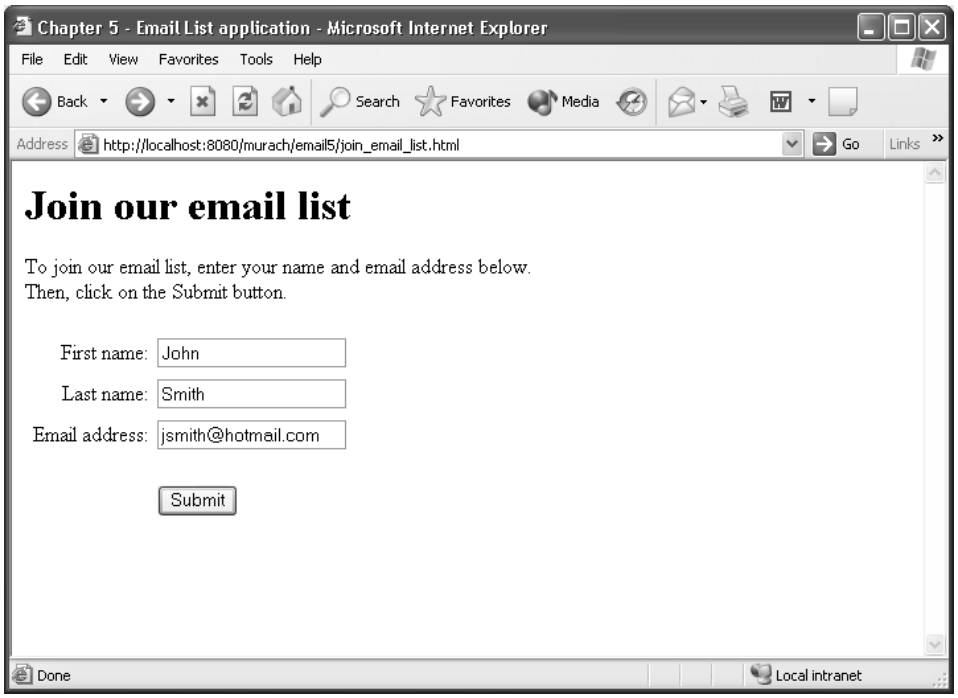
In this topic, you'll see how the Email List application that was presented in the last chapter works when it uses a servlet instead of a JSP. Once you get the general idea of how this works, you'll be ready to learn the specific skills that you need for developing servlets.

The user interface for the application

Figure 5-1 shows the two pages that make up the user interface for the Email List application. These pages are the same as the pages that are shown in the last chapter except that the second page uses a servlet instead of a JSP.

Like the last chapter, the first page is an HTML page that asks the user to enter a first name, last name, and email address. However, when the user clicks on the Submit button for this application, the HTML page calls a servlet instead of a JSP and passes the three user entries to the servlet. This is shown by the URL that's displayed in the browser for the second page.

The HTML page



The servlet page

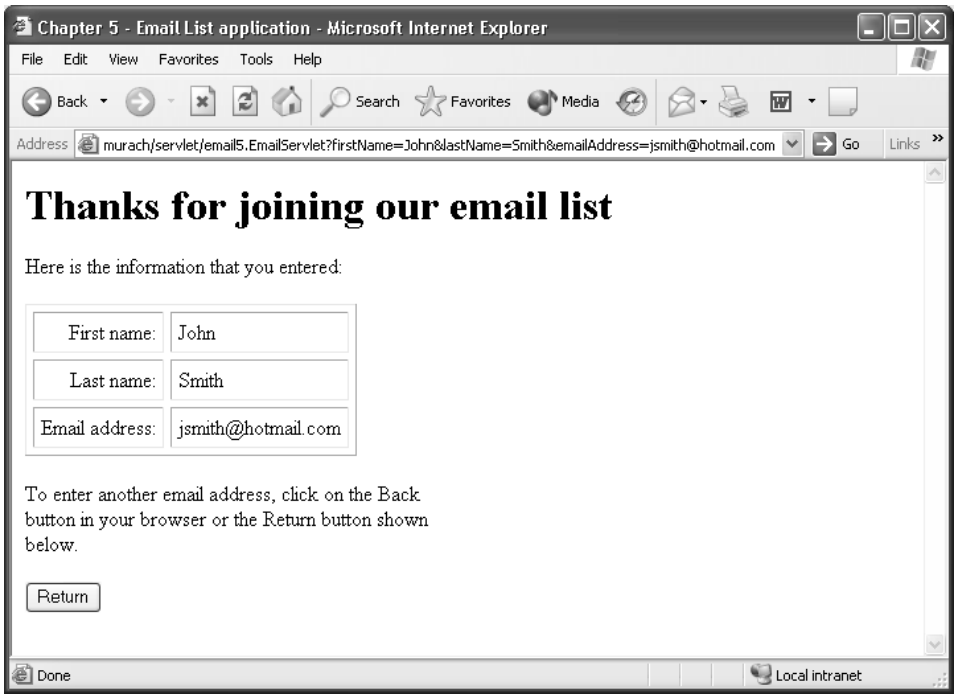


Figure 5-1 The user interface for the application

The code for the EmailServlet class

In chapter 1, you learned that a *servlet* is just a Java class that runs on a server. You also learned that a servlet for a web application extends the HttpServlet class. In figure 5-2, then, you can see that the EmailServlet class extends this class.

The first six statements for this servlet create the package for the servlet and import all the other classes that the servlet will need. First, the package statement puts the servlet in a package named email5. Then, the next three statements import some packages from the servlet API and from the core Java API. These packages are needed by all servlets. Finally, the last two statements import the User and UserIO classes from the business and data packages.

After the declaration for the class, the doGet method provides the code that's executed when a browser uses the Get method to request a servlet. This doGet method accepts two arguments that are passed to it from the web server: an HttpServletRequest object and an HttpServletResponse object. These objects are commonly referred to as the *request object* and the *response object*. In fact, the implicit request object that you learned about in the last chapter is actually an object of the HttpServletRequest class.

Within the doGet method, the first two statements perform tasks that are common to most servlets that return an HTML document. Here, the first statement sets the *content type* that will be returned to the browser. In this case, the content type is set so that the servlet returns an HTML document, but it's possible to return other content types. Then, the second statement returns a PrintWriter object that's used to return data to the browser later on.

After the first two statements of the doGet method, the shaded statements perform the same processing that was presented in the last chapter. First, the getParameter method of the request object returns the three parameters entered by the user. Then, a User object is created from these three parameters, and the UserIO class adds the User object to the specified file.

The last statement in the doGet method uses the println method of the PrintWriter object to return HTML to the browser. This is the same HTML that was used in the JSP in the last chapter. However, this HTML is more difficult to code and read now that it's coded as a string argument of the println method. That's why the next chapter will show how to remove this type of tedious coding from your servlets by combining the use of servlets with JSPs.

Within the println string, only the firstName, lastName, and emailAddress variables are displayed outside of quotation marks. Also, the quotation marks within the string, like the quotation marks around HTML attributes, are preceded by a backslash (\). Otherwise, the Java compiler would interpret those quotation marks as the end of the string. Since the quotation marks within the HTML statements aren't actually required, you can remove them to simplify the code, but the code is still clumsy.

The code for the EmailServlet class

```
package email5;

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

public class EmailServlet extends HttpServlet{

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                      throws IOException, ServletException{

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        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");

        out.println(
          "<!doctype html public \"-//W3C//DTD HTML 4.0 Transitional//EN\">\n"
        + "<html>\n"
        + "<head>\n"
        + "  <title>Chapter 5 - Email List application</title>\n"
        + "</head>\n"
        + "<body>\n"
        + "<h1>Thanks for joining our email list</h1>\n"
        + "<p>Here is the information that you entered:</p>\n"
        + "  <table cellspacing=\"5\" cellpadding=\"5\" border=\"1\">\n"
        + "  <tr><td align=\"right\">First name:</td>\n"
        + "      <td>" + firstName + "</td>\n"
        + "  </tr>\n"
        + "  <tr><td align=\"right\">Last name:</td>\n"
        + "      <td>" + lastName + "</td>\n"
        + "  </tr>\n"
        + "  <tr><td align=\"right\">Email address:</td>\n"
        + "      <td>" + emailAddress + "</td>\n"
        + "  </tr>\n"
        + "  </table>\n"
        + "<p>To enter another email address, click on the Back <br>\n"
        + "button in your browser or the Return button shown <br>\n"
        + "below.</p>\n"
        + "<form action=\"/murach/email5/join_email_list.html\" "
        + "      method=\"post\">\n"
        + "  <input type=\"submit\" value=\"Return\">\n"
        + "</form>\n"
        + "</body>\n"
        + "</html>\n");
    }
}
```

Figure 5-2 The code for the EmailServlet class

How to create a servlet

Now that you have a general idea of how servlets are coded, you're ready to learn some specific skills for creating a servlet. To start, you need to know how to begin coding the class for a servlet.

How to code a servlet

Figure 5-3 shows the basic structure for a typical servlet that performs some processing and returns an HTML document to the browser. You can use this basic structure for all the servlets you write. For now, that's all you need to get started, but you'll learn another way to structure servlets in the next chapter.

Since most servlets are stored in a package, the first statement in this servlet specifies the package for the servlet. This package must correspond to the directory that the servlet is saved in. In the next figure, you'll learn more about how this works.

The next three statements are the import statements that are required by all servlets. The javax.servlet.http package is required because it contains the HttpServletRequest and HttpServletResponse classes. The javax.servlet class is required because it contains the ServletException class. And the java.io class is required because it contains the IOException class.

After the first four statements, the class declaration provides the name for the servlet and indicates that it extends the HttpServlet class. Although in theory a servlet can extend the GenericServlet class, all servlets for web applications extend the HttpServlet class.

The doGet and doPost methods in this figure accept the same arguments and throw the same exceptions. Within these methods, you can use the methods of the request object to get incoming data, and you can use the methods of the response object to set outgoing data. In this structure, since the doPost method calls the doGet method, an HTTP request that uses the Post method will actually execute the doGet method of the servlet. This is a common programming practice that allows a servlet to use the same code to handle both the Get and Post methods of an HTTP request.

In the doGet method, the first statement calls the setContentType method of the response object. This sets the content type for the HTTP response that's returned to the browser to "text/html," which specifies that the servlet returns text or HTML. In chapter 15, you'll learn how to return other types of data.

The second statement in the doGet method obtains a PrintWriter object named out from the response object. This object can be used to return character data to the client. Once you have this object, you can use one println statement or a series of print and println statements to return HTML or other text to the browser. However, as you learned in the last figure, you must code a backslash before any quotation marks that don't start or end a string. Otherwise, the servlet won't compile properly.

The basic structure of a servlet class

```
package packageName;

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

public class ServletName extends HttpServlet{

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                      throws IOException, ServletException{

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        //business processing

        out.println("response as HTML");
    }

    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
                       throws IOException, ServletException {
        doGet(request, response);
    }

}
```

Description

- All web-based servlets extend the HttpServlet class. To extend this class, the servlet must import the javax.servlet, javax.servlet.http, and java.io packages.
- The doGet method overrides the doGet method of the HttpServlet class and processes all HTTP requests that use the Get method, and the doPost method overrides the doPost method of the HttpServlet class and processes all HTTP requests that use the Post method.
- The doGet and doPost methods use two objects that are passed to it by the web server: (1) the HttpServletRequest object, or the *request object*, and (2) the HttpServletResponse object, or the *response object*.
- The setContentType method of the response object sets the *content type* of the response that's returned to the browser. Then, the getWriter method of the response object returns a PrintWriter object that can be used to send HTML to the browser.
- Before you can create a PrintWriter object, you must set the content type. This allows the getWriter method to return a PrintWriter object that uses the proper content type.

Figure 5-3 How to code a servlet

How to save and compile a servlet

In chapter 4, you learned how to save and compile regular Java classes in Tomcat. Now, figure 5-4 shows you how to save and compile servlet classes.

Since the EmailServlet class contains a package statement that specifies a package named email5, it must be placed in the email5 subdirectory of the WEB-INF\classes directory that is subordinate to the document root directory. For the book applications, this root directory is webapps\murach.

This figure also presents four other directory paths that you can use to store your servlet classes. All of these paths store the servlet classes in the WEB-INF\classes directory that's subordinate to the document root directory. That can be your own document root directory or the default document root directory (ROOT) that's provided by Tomcat 4.0. Then, if you use packages, you continue the directory structure with one directory for each package level. If, for example, you use a class named EmailServlet in a package named business.email, the directory path for the servlet must be WEB-INF\classes\business\email.

If you're using TextPad to save and compile your servlets, you will need to set your classpath so it includes the directory that contains your class files as described in appendix A. Then, TextPad's Compile Java command will work properly.

Otherwise, you can use a DOS prompt to compile your packaged classes. To do that, you use the cd command to change the current directory to the WEB-INF\classes directory for your application. Then, you use the javac command to compile the class. When you do that, you can enter the javac command, followed by a space, followed by the package name, followed by a backslash (\), followed by the name of the Java class.

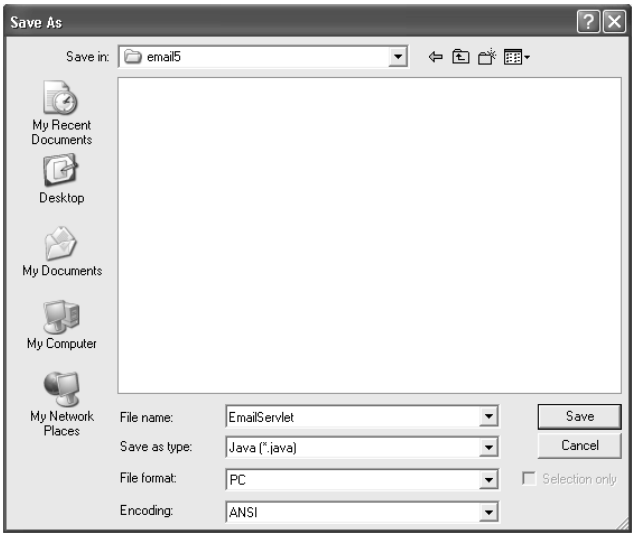
Where the EmailServlet class is saved

```
c:\tomcat\webapps\murach\WEB-INF\classes\email5
```

Other places to save your servlet classes

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

The TextPad dialog box for saving Java files



The DOS prompt window for compiling the EmailServlet class

Description

- Although you can save the source code (.java file) for a servlet in any directory, you must save the compiled class (.class file) in the \WEB-INF\classes subdirectory of the document root directory or in a subdirectory that corresponds to the Java package that the class is in.
- For simplicity, you can store your servlet source files in the same directory as the corresponding class files, but the source files for a production application are likely to be stored in other directories.
- To compile your servlets, you can use TextPad's Compile Java command, your IDE's compile command, or the javac command from a DOS prompt window.

Figure 5-4 How to save and compile a servlet

How to request a servlet

Now that you can code, save, and compile a servlet, you're ready to test the servlet by viewing it in a web browser. To do that, the web server and the servlet engine must be running. Then, you can request the servlet as shown in figure 5-5.

To test a servlet, you can enter a URL directly in a web browser. However, once you're done testing a servlet, you'll usually want to code a web page that requests the servlet. You can do that by coding a Form tag or an Anchor tag that specifies a URL that requests the servlet.

To request a servlet by entering the URL into a browser, you enter an absolute URL like the two examples in this figure. If you've read the last two chapters, you shouldn't have much trouble understanding how this works. The main difference is that you must enter "servlet" after the document root directory to indicate that you want to run a servlet. Then, you enter the package name for the servlet, followed by a period (or dot), followed by the name of the servlet.

The first example shows the URL for a servlet that's stored on a local web server in the email5 directory of the murach directory. The second example shows the URL for the servlet after it's deployed on the Internet server for *www.murach.com*.

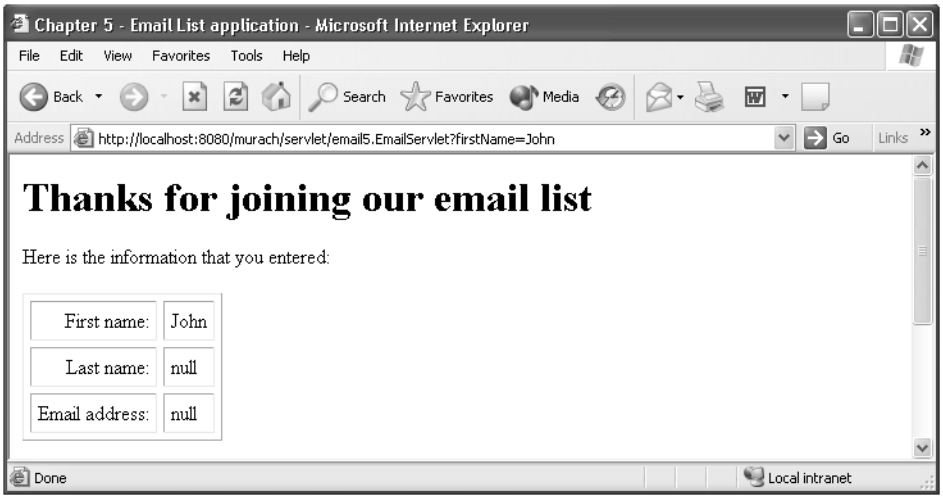
When you test a servlet, you will often want to pass parameters to it. To do that, you can add the parameters to the end of the URL. Here, the question mark after the servlet name indicates that one or more parameters will follow. Then, you can code the parameter name, the equals sign, and the parameter value for each parameter that is passed, and you can separate multiple parameters with ampersands (&).

If you omit a parameter that's required by the servlet, the getParameter method will return a null value for that parameter. In this figure, for example, only the first parameter value is added to the end of the URL. As a result, two of the values displayed by the servlet are null values.

To request a servlet from an HTML form, you use the Action attribute of the form to provide a path and filename that points to the servlet. This is illustrated by the last two examples in this figure. Here, the assumption is that the HTML page is in a subdirectory of the document root directory. As a result, the path specified in the Action attribute begins with two periods to navigate back one level to the root directory. Then, the word "servlet" specifies that you want to call a servlet. Last, the package and class names point to the class for the servlet.

When you use an HTML form to request a servlet, you can use the Method attribute to specify whether you want to use the Get method or the Post method. If you use a Get method to request a servlet from another page, any parameters that are passed to the servlet will be displayed in the browser at the end of the URL. But if you use the Post method to request a servlet, the parameters won't be displayed at the end of the URL.

How to request the EmailServlet class



The syntax for requesting a servlet

```
http://host:port/documentRoot/servlet/packageName.ServletName
```

Two URLS that request a servlet

```
http://localhost:8080/murach/servlet/email5.EmailServlet
http://www.murach.com/servlet/email5.EmailServlet
```

How to add parameters to a URL

```
EmailServlet?firstName=John&lastName=Smith
EmailServlet?firstName=John&lastName=Smith&emailAddress=jsmith@hotmail.com
```

Two Form tags that request a servlet

```
<form action="../servlet/email5.EmailServlet" method="get">
<form action="../servlet/email5.EmailServlet" method="post">
```

Description

- To request a servlet without using an HTML form, enter a URL that requests the servlet in the browser. If necessary, add parameters to the end of the URL.
- When you code or enter a URL that requests a servlet, 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 them.
- If an HTML form that uses the Get method requests a servlet, the URL and its parameters will be displayed by the browser. If an HTML form that uses the Post method requests a servlet, the URL will be displayed in the browser, but the parameters won't be.

Figure 5-5 How to request a servlet

Other skills for working with servlets

Now that you have a basic understanding of how to code and test a servlet, you're ready to learn some other skills for working with servlets. To start, you can learn about other methods that are available when you code servlets.

The methods of a servlet

Figure 5-6 presents some common methods of the HttpServlet class. When you code these methods, you need to understand that the servlet engine only creates one instance of a servlet. This usually occurs when the servlet engine starts or when the servlet is first requested. Then, each request for the servlet starts (or "spawns") a thread that can access that one instance of the servlet.

When the servlet engine creates the instance of the servlet, it calls the init method. Since this method is only called once, you can override it in your servlet to supply any necessary initialization code. In the next figure, you'll see an example of this.

After the servlet engine has created the one instance of the servlet, each request for that servlet spawns a thread that calls the service method of the servlet. This method checks the method that's specified in the HTTP request and calls the appropriate doGet or doPost method.

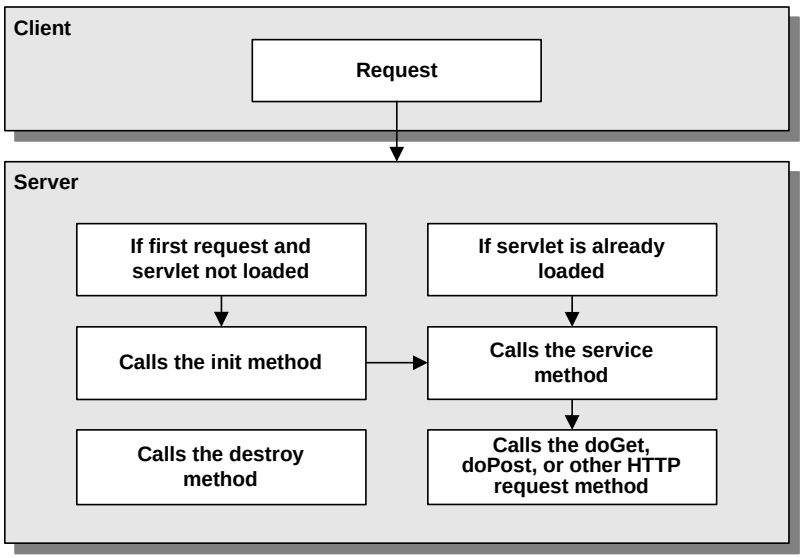
When you code servlets, you shouldn't override the service method. Instead, you should override the appropriate doGet or doPost methods. To handle a request that uses the Get method, for example, you can override the doGet method. If, on the other hand, you want to handle a request that uses the Post method, you can override the doPost method. To handle both types of requests, you can override both of them and have one call the other as shown in figure 5-3.

If a servlet has been idle for some time or if the servlet engine is shut down, the servlet engine unloads the instances of the servlets that it has created. Before unloading a servlet, though, it calls the destroy method of the servlet. If you want to provide some cleanup code, such as writing a variable to a file or closing a database connection, you can override this method. However, the destroy method may not be called if the server crashes. As a result, you shouldn't rely on it to execute any code that's critical to your application.

Five common methods of a servlet

```
public void init() throws ServletException{}

public void service(HttpServletRequest request,
                    HttpServletResponse response)
                    throws IOException, ServletException{}

public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
                  throws IOException, ServletException{}

public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
                   throws IOException, ServletException{}

public void destroy(){}
```

How the server handles a request for a servlet



The life cycle of a servlet

- A server loads and initializes the servlet by calling the init method.
- The servlet handles each browser request by calling the service method. This method then calls another method to handle the specific HTTP request type.
- The server removes the servlet by calling the destroy method. This occurs either when the servlet has been idle for some time or when the server is shutdown.

Description

- All the methods shown above are located in the abstract HttpServlet class. This means you can override these methods in your own servlets. However, you shouldn't need to override the service method. Rather, you should override a method like doGet or doPost to handle a specific HTTP request.

Figure 5-6 The methods of a servlet

How to code instance variables

Figure 5-7 shows how to code *instance variables* in a servlet. Since multiple threads access a single instance of a servlet, the instance variables of a servlet are used by all the threads. For that reason, you may need to synchronize the access to some instance variables if there's a chance that use by two or more threads at the same time could corrupt the data.

The code in this figure shows how to add an instance variable named accessCount to the EmailServlet. To initialize a variable like this, you can code its declaration in the init method. This will ensure that the variable is initialized when the instance of the servlet is first created.

If you shut down the server and restart it, though, the servlet will be destroyed and a new instance of the servlet will be created. As a result, any instance variables will be initialized again. If that's not what you want, you can write the value of the instance variable to a file so the data isn't lost when the servlet is destroyed.

In this example, the synchronized and this keywords prevent two threads from running the block of code that updates the accessCount instance variable at the same time. Then, after the thread updates the accessCount variable, its value is stored in a local variable so it can't be accessed by other threads. Later, this local variable is used by the println method to return the value to the browser. This is similar to the code that you saw in the scriptlet for the JSP in the last chapter.

Code that adds an instance variable to the EmailServlet class

```
public class EmailServlet extends HttpServlet{

    private int accessCount;

    public void init() throws ServletException{
        accessCount = 0;
    }

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                      throws IOException, ServletException{

        // missing code

        int localCount = 0;
        synchronized(this){
            accessCount++;
            localCount = accessCount;
        }

        // missing code

        out.println(
        // missing code
        + "<i>This page has been accessed " + localCount + " times.</i>"
        // missing code
        );
    }
}
```

Description

- An *instance variable* of a servlet belongs to the one instance of the servlet and is shared by any threads that request the servlet.
- You can initialize an instance variable in the init method.
- If you don't want two or more threads to modify an instance variable at the same time, you must synchronize the access to the instance variables.
- To synchronize access to a block of code, you can use the synchronized keyword and the this keyword.

Figure 5-7 How to code instance variables

How to code thread-safe servlets

A *thread-safe* servlet, like a thread-safe JSP, is one that works correctly even if more than one servlet thread is running at the same time. To code a thread-safe servlet, you not only have to synchronize the use of any instance variables that could be corrupted, but also any methods that could cause problems if they were used by two or more threads at the same time.

In figure 5-8, you can see how to limit the use of code to a single thread at three different levels. First, you can synchronize a block of code by using the synchronized and this keywords. Second, you can synchronize an entire method. Third, you can use the SingleThreadModel interface to limit the use of an entire servlet to a single thread. That way, you don't have to synchronize any of the code within the servlet.

As you code thread-safe servlets, you must remember that one thread has to wait while another thread is using a synchronized block of code, a synchronized method, or a single-thread servlet. Since that can affect the performance of a web application, your general goal should be to synchronize as little code as possible. As a result, synchronizing a block of code within a method is best for performance, synchronizing a method is next best, and coding single-thread servlets should usually be avoided.

But those decisions also depend on other factors like how long it takes to run a servlet or a method and how many users are likely to access a servlet at any given time. If the pages on your web site get accessed thousands of times each day, performance is obviously an issue. But if your web site gets only a few dozen visitors a day, performance may not be an issue at all.

A block of synchronized code

```
synchronized(this){
    accessCount++;
    if (accessCount == 1000){
        LogUtil.logToFile("We reached 1000 users on "
                          + new java.util.Date());
    }
}
```

A synchronized method

```
public static synchronized int addRecord(Connection connection, User user)
                                         throws SQLException{
    String query =
           "INSERT INTO User " +
           "(EmailAddress, FirstName, LastName) " +
           "VALUES ('" + SQLUtil.encode(user.getEmailAddress()) + "', " +
                    "'" + SQLUtil.encode(user.getFirstName()) + "', " +
                    "'" + SQLUtil.encode(user.getLastName()) + "')";
    Statement statement = connection.createStatement();
    int status = statement.executeUpdate(query);
    statement.close();
    return status;
}
```

A servlet that prevents multiple thread access

```
public class EmailServlet extends HttpServlet implements SingleThreadModel{…
}
```

Description

- A *thread-safe* servlet is one that works correctly when more than one thread is running at the same time.
- To synchronize access to a method or a block of code, you can use the synchronized keyword. Then, only one thread at a time can access the code in the block or the method.
- To prevent multiple threads from accessing the code in a servlet, you can implement the SingleThreadModel interface. This is a tagging interface that prevents two threads from accessing the servlet's service method at the same time.

Note

- A tagging interface is one that has no methods.

Figure 5-8 How to code thread-safe servlets

How to debug servlets

When you develop servlets, you will encounter errors. That's why this topic gives you some ideas on how to debug servlets when you're using a text editor to develop your servlets. If you're using an IDE, though, it may provide advanced debugging tools that let you step through code and monitor variables so you won't the need to use the techniques that follow.

Common servlet problems

Figure 5-9 lists four common problems that can occur when you're working with servlets. Then, it lists some possible solutions for each of these problems.

If your servlet won't compile, the error message that's displayed by the compiler should give you an idea of why the servlet won't compile. If the compiler can't find a class that's in one of the Java APIs, for example, you may need to install the API. If the compiler can't locate your custom classes, you may need to modify your classpath. And if the compiler has a problem locating a package, your package statement for the class might not correspond with the directory that contains the class.

If the servlet compiles but won't run, it may be because the servlet engine isn't running. To solve this problem, of course, you can start the servlet engine. However, if the servlet engine is already running, you should double-check the URL to make sure that it's pointing to the correct host, path, and package for the servlet. A common mistake, for example, is to forget to include the package name when specifying a URL for a servlet.

If you make changes to a servlet and the changes aren't apparent when you test it, it may be because the servlet engine hasn't reloaded the modified class. Then, if you're using Tomcat in a stand-alone development environment, you can either make sure that servlet reloading is turned on as explained in chapter 2. Or, you can shutdown Tomcat and restart it so the changed servlet will be reloaded the next time that it's requested.

If the HTML response page doesn't look right when it's rendered by the browser, the servlet is probably sending bad HTML to the browser. To fix this problem, you can use the Source command (for Internet Explorer) or the Page Source command (for Netscape) to view the HTML that has been returned to the browser. Then, you can identify the problem and modify the servlet to fix it.

Common servlet problems

Problem	Possible solutions
The servlet won't compile	Make sure the compiler has access to the JAR files for all necessary APIs.
	Make sure the classpath is pointing to the directory that contains your user-defined packages.
	Make sure the class is in the correct directory with the correct package statement.
The servlet won't run	Make sure the web server is running.
	Make sure you're using the correct URL.
The changes aren't showing up	Make sure servlet reloading is on, or shutdown and startup the server so it reloads the class that you modified.
The HTML page doesn't look right	Select the Source or Page Source command from your browser's View menu to view the HTML code. Then, you can read through the HTML code to identify the problem, and you can fix the problem in the servlet.

Note

- Appendix A shows how to install the Java APIs and how to set the classpath.

Figure 5-9 Common servlet problems

How to print debugging data to the console

If you're using Tomcat in a stand-alone development environment, you can print debugging messages to the console for the servlet engine as shown in figure 5-10. To do that, you can use the println method of the System.out and System.err objects. You can use these message to help track the methods that are executed or the changes to variables.

If you aren't using Tomcat in a stand-alone environment, you might not be able to view the debugging messages in the console. In some cases, though, you may be able to make them appear in the console by starting the servlet engine from a command line. In other cases, the println statements may automatically be written to a text file that you can view. Otherwise, you'll need to print this data to a log file as described in the next figure.

When you use println statements to check the value of a variable, you'll often want to include the name of the class and the name of the variable. That way, your messages will be easier to understand. This also makes it easier to find and remove the println statements once the error is debugged.

How Tomcat displays println statements

Code that prints debugging data to the console

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
                  throws IOException, ServletException{
    // code
    String emailAddress = request.getParameter("emailAddress");
    System.out.println("EmailServlet emailAddress: " + emailAddress);
    // code
}
```

Description

- When you're testing an application on your local system, you may be able to use the println method of the System.out or System.err objects to display debugging messages on the console for the servlet engine.
- When you use debugging messages to display variable values, it's a good practice to include the class name and variable name so the messages are easy to interpret.

Figure 5-10 How to print debugging data to the console

How to write debugging data to a log file

If you can't print debugging data to the console, you can print debugging data to a *log file* as shown in figure 5-11. Although each servlet engine uses log files a little differently, you should be able to use these log methods with any servlet engine. However, you may need to check the documentation for your servlet engine to see how it works with log files.

To write data to a log file, you can use the two log methods of the HttpServlet class. If you just want to write a message to a log file, you can use the first log method. But if you want to write a message to the log file along with the stack trace for an exception, you can use the second log method. A *stack trace* is a series of messages that presents the chain of method calls that precede the current method.

The code in this figure uses the first log method to display the value for the emailAddress variable of the EmailServlet class. Then, it uses the second log method to print a message and a stack trace for an IOException. The data that's printed by these two log methods is shown in the TextPad window.

Tomcat 4.0 stores all log files in its logs directory. Within this directory, Tomcat stores several types of log files with one file of each type for each date. In the TextPad window, you can see the contents of the localhost_log.2002-10-15.txt file. This log file contains the value of the emailAddress variable as well as the stack trace for an IOException.

Often, the servlet engine automatically writes other information in the same log file that the log methods of the HttpServlet class use. In that case, you can write your debugging information to a separate text file to make it easier to view your debugging messages. To do that, you can create your own class that writes error messages to a file.

To illustrate, the CD that comes with this book includes a LogUtil class in the util package that contains log methods that work like the log methods shown in this figure. However, you can easily modify this class to specify a name and location for the log file. Then, you can use the log methods in this class to write your debugging information to the file.

Two methods of the HttpServlet class used to log errors

Method	Description
`log(String message)`	Writes the specified message to the server's error log.
`log(String message, Throwable t)`	Writes the specified message and stack trace for the exception to the server's error log.

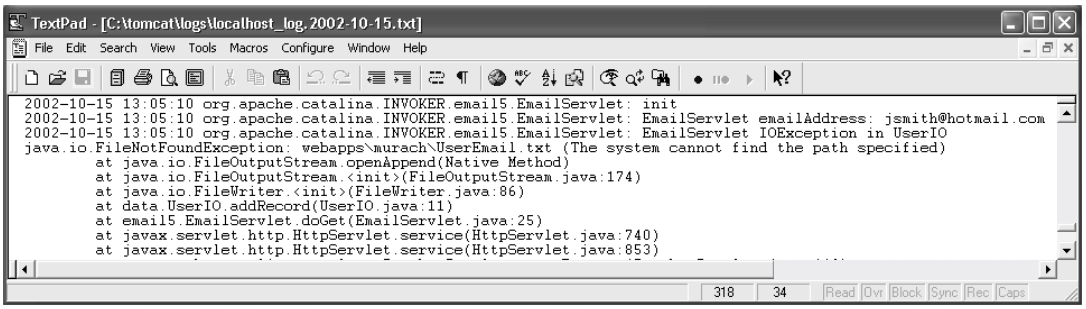
Servlet code that prints data to a server specific log file

```
String emailAddress = request.getParameter("emailAddress");
log("EmailServlet emailAddress: " + emailAddress);
User user = new User(firstName, lastName, emailAddress);
try{
    UserIO.addRecord(user, file);
}
catch(IOException ioe){
    log("EmailServlet IOException in UserIO", ioe);
}
```

The location of log files in Tomcat 4.0

```
\tomcat\logs
```

A typical log file



Description

- You can use the log methods of the HttpServlet class to write debugging information to a *log file*. Tomcat 4.0 stores its log files in a directory named logs.
- The name and location of the log files for a servlet engine may vary depending on the servlet engine. To find the name and location of your log files, check the documentation for your servlet engine.
- A *stack trace* is the chain of method calls for any statement that calls a method.

Figure 5-11 How to write debugging data to a log file

Perspective

The goal of this chapter has been to teach you the basics of coding, saving, and testing servlets. So at this point, you should be able to develop simple, but practical, servlets of your own. In addition, you should have a basic understanding of how servlets are executed.

Note, however, that you usually don't use servlets to send the HTML code back to the browser as shown in this chapter. Instead, you structure your web applications so servlets do the processing that's required and JSPs send the HTML code back to the browser. In that way, you combine the best features of servlets with the best features of JSPs, and that's what you'll learn how to do in the next chapter.

Summary

- A *servlet* is a Java class that runs on a server, and a servlet for a web application extends the HttpServlet class.
- When you write servlets, you override the doGet and doPost methods to provide the processing that's required. These methods receive the *request object* and the *response object* that are passed to them by the server.
- After you use the setContentType method of the response object to set the *content type* of the response that's returned to the browser, you use the getWriter method to create a PrintWriter object. Then, you can use the println and print methods of that object to send HTML back to the browser.
- The class files for servlets must be stored in the WEB-INF\classes directory of an application or in a subdirectory that corresponds to a package name.
- To request a servlet from a URL, you include the word "servlet" after the document root directory in the path. This is followed by the package name, a dot, and the servlet name.
- You can override the init method of a servlet to initialize its instance variables. These variables are then available to all of the threads that are spawned for the one instance of the servlet.
- When you code a *thread-safe* servlet, you prevent two or more users from accessing the same block of code at the same time.
- To print debugging data to the server console, you can use the println method of the System.out or System.err object. An alternative is to use the log methods of the HttpServlet class to write debugging data to a *log file*.

Terms

servlet
request object
response object
content type
instance variable
thread-safe
log file
stack trace

Objectives

- Code and test servlets that require any of the features presented in this chapter.
- Provide debugging data for a servlet by writing messages to either the console or a log file.
- Describe the directory structure that must be used for servlet classes.
- Describe the difference between the URL for a JSP and the URL for a servlet.
- Describe the use of the init, doGet, and doPost methods in a servlet.
- Describe the execution of a servlet, and explain its effect on local and instance variables.
- Explain what is meant by a thread-safe servlet and describe what you have to do to develop one.

Exercise 5-1 Modify the EmailServlet class

1. Enter a URL in your browser that requests the EmailServlet class that's located in the murach/WEB-INF/classes/email5 directory and sends two parameters to it.
2. Run the HTML document named join_email_list.html in the email5 directory so it accesses and runs the EmailServlet class.
3. Add the accessCount instance variable to the EmailServlet class as described in figure 5-7. Then, run the servlet from the HTML document. This will test whether servlet reloading has been turned on as described in chapter 2. If it isn't on, you will have to stop and restart Tomcat before the changes to the servlet show up. (You may have to refresh your browser for the changes to take effect.)
4. Modify the HTML document so it uses the Post method instead of the Get method, and modify the servlet so it works properly.
5. Print a debugging message to the console that shows the value of the accessCount variable. Then, run the servlet two or more times to see how this message appears in the console.
6. Repeat step 5, but use a log file this time.

Exercise 5-2 Create a new servlet

In this exercise, you'll modify the HTML document for the Email List application, and you'll create a new servlet that responds to the HTML document. This is comparable to what you did for exercise 4-2, but the details are repeated here.

1. Modify the HTML document named join_email_list.html that's in the email5 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 servlet named MusicChoicesServlet.
2. Create a new servlet named MusicChoicesServlet that responds to the changed HTML document. This servlet should respond with an H1 line that looks 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 servlet by running them. Note the parameter list that is passed to the servlet by the HTML document. Then, test the new servlet by using a URL that includes a parameter list.