5

How to develop servlets

Chapter 2 showed how to use the MVC pattern to structure a web application that uses servlets and JSPs. Now, this chapter presents the skills that you need to develop servlets within the MVC pattern. When you complete this chapter, you should be able to code the servlets for web applications of your own.

Before you start this chapter, though, take a few minutes to review the figures in chapter 2 that present the Email List application. That will refresh your memory about how the HTML, JSP, servlet, and web.xml files are related and how servlets fit into the MVC pattern.

How to create and map a servlet

There are two steps to creating a servlet. You must code the class for the servlet, and you must map that class to a URL. Prior to the servlet 3.0 specification (Tomcat 7.0), you had to use the web.xml to map a servlet to a URL. With the servlet 3.0 specification and later, you can use the @WebServlet annotation to map a servlet to one or more URL patterns.

How to create a servlet

Figure 5-1 shows the basic structure for a servlet that handles both HTTP GET and POST requests. For now, you can use this basic structure to get started with servlets. As you progress through this chapter, you'll learn more about working with servlets.

The first statement for this servlet specifies the package for the servlet's class. This package must correspond to the directory that contains the servlet. Then, the next six statements import the classes that are required by this servlet. Most of these classes are required by all servlets. The only class that isn't required by all servlets is the PrintWriter class, which is only necessary for a servlet that needs to send text like HTML or XML to the client.

After the first four statements, the class declaration provides the name for the class and indicates that it extends the HttpServlet class. In theory, a servlet can extend the GenericServlet class. In practice, however, all servlets 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 example, the doGet method calls the doPost method. As a result, an HTTP GET request executes the doPost method of the servlet. This allows a servlet to use the same code to handle both the GET and POST methods of an HTTP request. Although this is a common practice, it's better to use the doGet and doPost methods only to execute code that's appropriate for the corresponding method of the HTTP request.

The second statement in the doGet method calls the getWriter method of the response object to get a PrintWriter object named out. Once you get 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 as shown by the third statement. Since this statement can throw an IOException, it's enclosed in a try block.

The last statement closes and flushes the output stream and releases any resources that are being used by the PrintWriter object. Since you always want to execute this statement, even if an exception is encountered, it's enclosed in a finally block.

A simple servlet that returns HTML

```
package murach.email;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class TestServlet extends HttpServlet {

    @Override
    protected void doPost(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        try {
            out.println("<h1>HTML from servlet</h1>");
        }
        finally {
            out.close();
        }
    }

    @Override
    protected void doGet(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {

        doPost(request, response);
    }
}
```

Description

- In practice, all servlets extend the HttpServlet class. To extend this class, the servlet must import some of the classes in the java.io, javax.servlet, and javax.servlet.http 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 5-1 How to code a servlet

How to map a servlet with the web.xml file

Before you can request a servlet, you need to map the servlet to a URL. One way to do that is to use the web.xml file for the application as shown in this figure. The advantage of this approach is that it works with all versions of Tomcat, new and old. However, if you're using the 3.0 servlet specification (Tomcat 7.0) or later, you may find it easier to use annotations to map servlets as shown in the next figure.

In figure 5-2, the web.xml file contains XML elements that map two servlets. The first servlet element in this figure declares a name that refers to the EmailListServlet class that's shown in the previous figure. Here, the servlet-name element provides a unique name for the class. This name is used internally by the web.xml file. Then, the servlet-class element uses a fully-qualified name to identify the class for the servlet. In this example, the servlet-name entry is the same as the name for the class, but it isn't qualified by the package name. This is a common convention for naming a servlet. However, if the same servlet name is used in two or more packages, you can use servlet elements to specify a unique name for each servlet.

The second servlet element uses the same naming convention as the first servlet element. It declares an internal name of TestServlet for a servlet named TestServlet that's stored in the murach.email package.

The first servlet-mapping element maps the EmailListServlet to a single URL that's available from the root directory of the application. As a result, the user is able to request this servlet by specifying a URL pattern like this one:

```
http://localhost:8080/ch05email/emailList
```

Note that this URL removes the word *Servlet* from the end of the servlet name. I have used this convention throughout this book because it shortens the URL and hides the fact that this application uses servlets from the user.

The second servlet-mapping element uses a wildcard character (*) to map the EmailListServlet to any URL that resides within the email directory. This allows this servlet to be requested by multiple URLs. For example, you could request this servlet with this URL:

```
http://localhost:8080/ch05email/email/add
```

Or, you could request this servlet with this URL:

```
http://localhost:8080/ch05email/email/addToList
```

It's important to note that this servlet mapping works even if the email directory is a virtual directory that doesn't actually exist on the server.

The third servlet-mapping element maps the TestServlet to a single URL. This element uses the same naming convention as the first servlet-mapping element.

If you have any trouble working with the web.xml file, you can review the skills for working with the web.xml file that are presented later in this chapter. Or, if necessary, you can get more information about working with XML from our Java book, *Murach's Java Programming*.

XML tags that add servlet mapping to the web.xml file

```
<!-- the definitions for the servlets -->
<servlet>
    <servlet-name>EmailListServlet</servlet-name>
    <servlet-class>murach.email.EmailListServlet</servlet-class>
</servlet>
<servlet>
    <servlet-name>TestServlet</servlet-name>
    <servlet-class>murach.email.TestServlet</servlet-class>
</servlet>

<!-- the mapping for the servlets -->
<servlet-mapping>
    <servlet-name>EmailListServlet</servlet-name>
    <url-pattern>/emailList</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>EmailListServlet</servlet-name>
    <url-pattern>/email/*</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>TestServlet</servlet-name>
    <url-pattern>/test</url-pattern>
</servlet-mapping>
```

XML elements for working with servlet mapping

Element	Description
`<servlet-class>`	Specifies the class for the servlet. Note that this element includes the package and name for the class but not the .class extension.
`<servlet-name>`	Specifies a unique name for the servlet that's used to identify the servlet within the web.xml file. This element is required for both the servlet element and the servlet-mapping element and maps each servlet-mapping element to a servlet element.
`<url-pattern>`	Specifies the URL or URLs that are mapped to the specified servlet. This pattern must begin with a front slash, but the URL pattern can specify a virtual directory or file that doesn't actually exist.

Some URL pattern examples

URL pattern	Description
`/emailList`	Specifies the emailList URL in the root directory of the application.
`/email/*`	Specifies any URL in the email directory.
`/email/add`	Specifies the add URL in the email directory.

Description

- Before you can request a servlet, you should use the application's web.xml file to map a servlet to a URL pattern.

Figure 5-2 How to map a servlet with the web.xml file

How to map a servlet with an annotation

As mentioned earlier, if you're using the 3.0 servlet specification (Tomcat 7.0) or later, you can use annotations to map servlets as shown in figure 5-3. The advantage of this approach is that it requires less code. Also, some developers find it easier to maintain the servlet mapping when it's stored in the same file as the source code for the servlet.

If you want to use annotations to map your servlets, you can begin by adding an import statement for the WebServlet class. Then, you can code the @WebServlet annotation just before the class declaration for the servlet. To map a servlet to a single URL, you can code the URL within the parentheses that follow the @WebServlet annotation as shown in the first example.

However, if you want to map a servlet to multiple URLs, you can use the urlPatterns attribute of the @WebServlet annotation as shown in the second example. Here, you code an equals sign (=) followed by a set of braces ({}). Then, within the braces, you code two or more URLs, separating each URL with a comma. If you want, you can use the wildcard character (*) to map a servlet to any URL that resides within a directory. This works as described in the previous figure.

By default, the internal name that's used for the servlet is the same as the class name of the servlet, which usually works fine. However, if this leads to a naming conflict, you can use the name attribute of the @WebServlet annotation to specify a unique internal name for the servlet. For instance, if you have a servlet class named TestServlet that's in two different packages, you can use the name attribute to specify a unique internal name for each servlet. In the third example, for instance, the annotation specifies a name of MurachTestServlet for the TestServlet class.

Typically, you use either the web.xml file or @WebServlet annotations to map the servlets in an application. As a result, you don't usually need to worry about which technique takes precedence. However, if you use both techniques to map a servlet name to the same URL, the mapping in the web.xml file overrides the mapping in the annotation.

A servlet that uses an annotation to map itself to a URL

```
package murach.email;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/test")
public class TestServlet extends HttpServlet {
...
...
```

How to map a servlet to multiple URLs

```
@WebServlet(urlPatterns={"/emailList", "/email/*"})
```

How to specify a name for the servlet

```
@WebServlet(name="MurachTestServlet", urlPatterns={"/test"})
```

Description

- With the servlet 3.0 specification (Tomcat 7.0) and later, you can use the @WebServlet annotation to map a servlet to one or more URL patterns.
- To map a servlet to a single URL pattern, code the URL pattern in the parentheses that follow the @WebServlet annotation.
- To map a servlet to multiple URL patterns, code the urlPatterns attribute within the parentheses that follow the @WebServlet annotation. Then, code an equals sign followed by a set of braces ({}). Within the braces, you can code multiple URL patterns, separating each pattern with a comma.
- If you encounter a naming conflict, you can use the name attribute of the @WebServlet annotation to specify a unique mapping name for the servlet.
- If you use both the web.xml and the @WebServlet annotation to map the same servlet name to one or more URLs, the mapping in the web.xml file overrides the mapping in the annotation.

Figure 5-3 How to map a servlet with an annotation

How to request a servlet

After you create and map a servlet, you can request the servlet to test it. When you do that, you can use an HTTP GET or POST request.

How to use the HTTP GET method

Figure 5-4 shows several ways to request a servlet. To do that, you typically code a form tag or an anchor (a) tag that requests the servlet as shown by the examples in this figure. However, if the web server and servlet engine are running, you can also enter a URL directly into the browser.

When you test a servlet, you often need to pass parameters to it. To do that, you can append the parameters to the end of the URL as shown by the first example. Here, the question mark after the servlet's URL 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 (&).

The browser shown in this figure contains a URL that calls a servlet and passes three parameters named action, firstName, and lastName. Since this request doesn't include a parameter for the email address, the email address isn't shown on the resulting web page.

There are three ways you can request a servlet. First, you can enter its URL into a browser as shown by the second set of examples. Here, the first URL requests a servlet that's mapped to the emailList directory that's running on a local server using port 8080 in an app named ch05email. Then, the second URL requests a servlet that's mapped to the email/list directory that's running on the web server for www.murach.com using the default port, port 80.

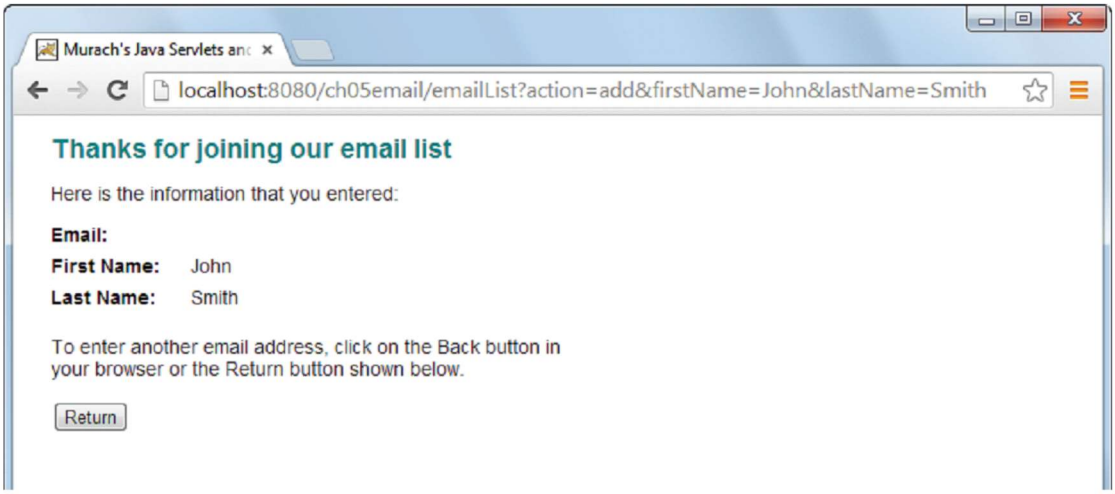
Second, you can use a form tag to request a servlet. When you use this technique, you code the action attribute of the form to provide a path and filename to the servlet's URL. In this figure, the first form tag requests a servlet that's mapped to the emailList subdirectory of the current directory. However, if the servlet is stored in a different directory, you can specify a relative or absolute path for the action attribute.

When you use a form tag to request a URL, you can use the method attribute to specify the HTTP method that's used for the request. By default, the form tag uses the GET method. However, you can explicitly specify the GET method if you want. In this figure, both of the form tags use the GET method. In the next figure, you'll learn when and how to specify the POST method.

When you use the GET method to request a servlet, any parameters that are passed to the servlet are displayed in the browser's URL address. In this figure, for example, the browser's URL includes three parameters. In the next figure, you'll learn that the POST method doesn't display the parameters in the URL for the browser.

Third, you can use an anchor tag to call a servlet. When you use an anchor tag, it always uses the HTTP GET method, and you can append parameters to the end of the URL. In this figure, the anchor tag appends one parameter to the URL.

A URL that's requested with the HTTP GET method



How to append parameters to a URL

```
emailList?action=add
emailList?firstName=John&lastName=Smith
```

Three ways to append parameters to a GET request

Enter the URL into the browser's address bar

```
http://localhost:8080/ch05email/emailList?action=add&firstName=John
http://www.murach.com/email/list?action=add&firstName=John
```

Code a form that uses the GET method

```
<form action="emailList">
<form action="emailList" method="get">
```

Code an anchor tag

```
<a href="emailList?action=join">Display Email Entry Test</a>
```

Description

- To append a parameter to a URL, code a question mark at the end of the URL followed by the name of the parameter, an equals sign, and the value of the parameter.
- To append multiple parameters to a URL, use ampersands (&) to separate the parameters.
- If you enter a URL into the browser's address bar, the browser uses the HTTP GET method.
- When you code a form, it uses the HTTP GET method by default, but you can specify this method for clarity if you want. When you use the HTTP GET method with a form, any parameters within the form are automatically appended to the URL displayed by the browser.
- When you code an anchor tag, you can append parameters to it. This tag always uses the HTTP GET method.
- To process an HTTP GET request, a servlet must implement the doGet method.

Figure 5-4 How to request a URL with the HTTP GET method

How to use the HTTP POST method

When you use a form tag to request a URL, you can use the HTTP POST method. To do that, you use the method attribute to specify the POST method as shown in figure 5-5. Then, the request uses the POST method, and browser doesn't display the parameters for the request in its URL.

When to use the HTTP GET and POST methods

So, when should you use the HTTP GET method and when should you use the POST method? In general, you should use the GET method when you want to *get* (read) data from the server. Similarly, you should use the POST method when you want to *post* (write) data to the server.

When you use the GET method, you need to make sure that the page can be executed multiple times without causing any problems. If, for example, the servlet just reads data from the server and displays it to the user, there's no harm in executing the servlet multiple times. If, for example, the user clicks the Refresh button, the browser requests the URL again, and this doesn't cause any problems.

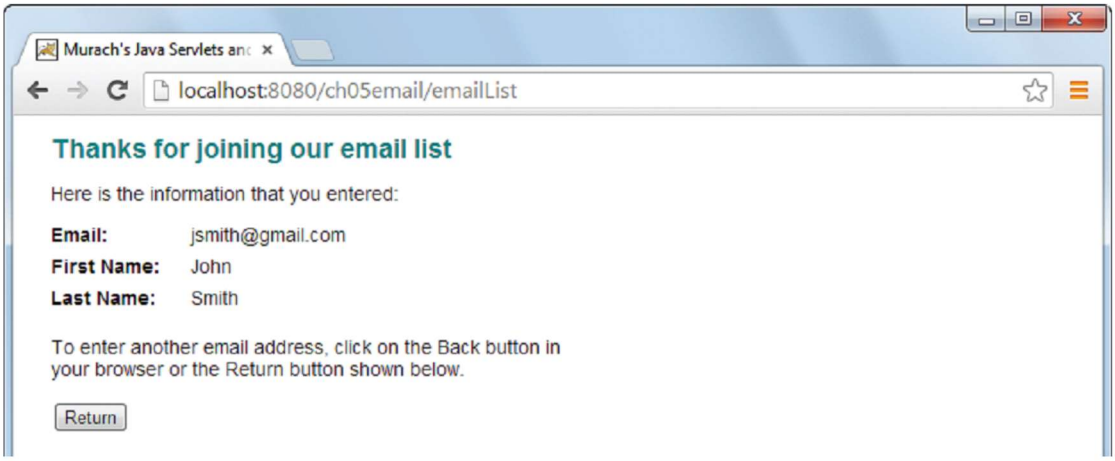
However, if the servlet writes or sends data to the server, you typically don't want the user to execute the servlet multiple times. As a result, it makes more sense to use the POST method. Then, if the user clicks the Refresh button, the browser displays a dialog box like the one shown in this figure that warns the user that the data is about to be submitted again. At that point, the user can click on the Cancel button to cancel the request.

There are also a few other reasons to use 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. Second, since the POST method prevents the web browser from including parameters in a bookmark for a page, you'll want to use it if you don't want the parameters to be included in the bookmark. Third, if your parameters contain more than 4 KB of data, the GET method won't work so you'll need to use the POST method.

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 are sent to the page.

When you request a servlet by using an anchor tag or by entering a URL into a browser, remember that these techniques automatically use the HTTP GET method. As a result, a servlet can process these requests only if it has implemented the doGet method. If you attempt to use one of these methods with a servlet that doesn't implement the doGet method, the server returns an error message that indicates that the HTTP GET method is not supported by the specified URL. Conversely, if you use the HTTP POST method to request a servlet that doesn't implement the doPost method, the server returns an error message that indicates that the HTTP POST method is not supported by the specified URL.

A URL that's requested with the HTTP POST method



A Form tag that uses the POST method

```
<form action="emailList" method="post">
```

Use the GET method when...

- The request reads data from the server.
- The request can be executed multiple times without causing any problems.

Use the POST method when...

- The request writes data to the server.
- Executing the request multiple times may cause problems.
- You don't want to include the parameters in the URL for security reasons.
- You don't want users to be able to include parameters when they bookmark a page.
- You need to transfer more than 4 KB of data.

A typical browser dialog that's displayed if the user tries to refresh a post

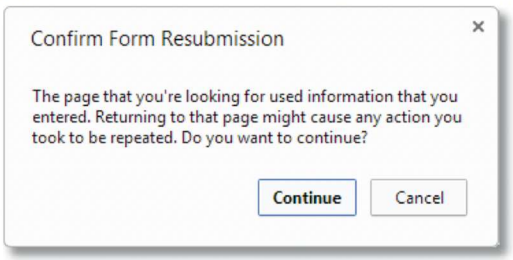


Figure 5-5 How to request a URL with the HTTP POST method

Skills for working with servlets

In the previous figures, you learned how to create a servlet, map it to a URL, and request it. Now, you'll learn some skills for processing the request.

How to get the values of the parameters

Figure 5-6 summarizes two of the methods available from the HttpServletRequest object, which is typically named request. This object is available to both the doGet and doPost methods of a servlet, and you can use the methods of this object to get the values of the parameters of the request.

To get the text that a user has entered into a text box, you can call the getParameter method from the request object and specify the name of the text box. In this figure, for instance, the first example gets the text that's stored in the text box that has a name attribute of "firstName". You can use a similar technique to work with other controls. For example, you can use this technique to get the selected value from a combo box or a group of radio buttons.

For checkboxes that have a value attribute, the getParameter method returns that value if the checkbox is selected and a null value if it isn't. However, if the checkbox doesn't have a value attribute, the getParameter method returns an "on" value if the checkbox is selected and a null value if it isn't. As a result, you can determine whether the user has selected a check box by checking whether its value is null as shown in the second example.

To retrieve multiple values for one parameter name, you can use the getParameterValues method as illustrated by the third example. This 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 as shown in the third example.

Two methods available from the HttpServletRequest 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 or JSP file.
`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.

Servlet code that gets text from a text box

```
String firstName = request.getParameter("firstName");
```

Servlet code that determines if a check box is checked

```
// returns the value or "on" if checked, null otherwise.
String rockCheckBox = request.getParameter("rock");
if (rockCheckBox != null)
{
    // rock music was checked
}
```

Servlet code that reads and processes multiple values from a list box

```
// returns the values of the items selected in a list box.
String[] selectedCountries = request.getParameterValues("country");
for (String country : selectedCountries)
{
    // code that processes each country
}
```

Description

- Within a servlet, the doGet and doPost methods both provide an HttpServletRequest object. This object is typically named request.
- You can use the methods of the request object to get the values of the parameters that are stored in the request.

Figure 5-6 How to get the values of the parameters

How to get the real path for a file

When you work with servlets, you typically use relative paths to refer to files that are available within the web application. Sometimes, though, you need to get the real path for one of these files.

To do that, you can use the technique shown in figure 5-7. First, you call the getServletContext method to get a ServletContext object. Then, you call the getRealPath method of the ServletContext object to return the real path for the specified file. When you use the getRealPath method, a front slash navigates to the root directory for the current application, so

```
getRealPath("/EmailList.txt")
```

specifies the EmailList.txt file in the current application's root directory.

In this figure, the first example gets the real path for a file named EmailList.txt that's stored in the WEB-INF subdirectory of the application. Here, the getRealPath method returns a string for an absolute path to this file. If, for example, this method is used in the ch05email application that's in the NetBeans directory that's used for our downloadable applications, the getRealPath method returns the path shown in this figure.

The second example shows a more concise way to code the first example. Here, the code doesn't supply a name for the ServletContext object. Instead, it calls the getRealPath method directly from the getServletContext method. Since this technique calls one method from another method, it's known as *method chaining*. Often, this technique results in code that is shorter and easier to read. The only disadvantage is that it doesn't create a variable name for the ServletContext object that you can use multiple times in your code. But, if your code only uses the ServletContext object once, it usually makes sense to use method chaining.

If you store a file in a directory that's web accessible, such as the root directory for a web application, any user who enters the correct URL can access the file. However, the WEB-INF directory and its subdirectories aren't directly web-accessible. As a result, if you want to keep a file private, you can store it in the WEB-INF directory or one of its subdirectories. Or, you can restrict access to the file or directory as described in chapter 16.

If you use the getRealPath method, you should be aware that it may work differently on different systems. For example, with Tomcat 8, you must start the path parameter with a front slash. However, with Tomcat 7, this front slash isn't required. Similarly, the getRealPath method fails and returns a null value if you are attempting to get a path to a resource that's inside a WAR file, or if you are trying to get a path to a resource that's located in or on a virtual file system.

A method of the GenericServlet class

Method	Description
`getServletContext()`	Returns a ServletContext object that contains information about the application's context.

A method of the ServletContext class for working with paths

Method	Description
`getRealPath(String path)`	Returns a String object for the absolute path of the specified relative path.

Code that gets the absolute path for a file

```
ServletContext sc = this.getServletContext();
String path = sc.getRealPath("/WEB-INF/EmailList.txt");
```

A more concise way to get the absolute path for a file

```
String path = this.getServletContext()
        .getRealPath("/WEB-INF/EmailList.txt");
```

A possible value for the real path variable

```
C:\murach\servlet_and_jsp\netbeans\book_apps\ch05email\build\web\WEB-INF\
EmailList.txt
```

Description

- All servlets inherit the GenericServlet class. As a result, the getServletContext method is available to all servlets.
- In addition to working with relative paths as described here, you can use the ServletContext object to read global initialization parameters, to work with global variables, and to write data to log files. You'll learn more about these skills as you progress through this book.

Figure 5-7 How to get the real path for a file

How to get and set request attributes

To store any object in the request object, you can use the setAttribute method as shown in figure 5-8. Here, the first example creates a User object named user and stores it in the request object with a name of "user".

Once you store an object in the request object, you can use the getAttribute method to retrieve the object. In this figure, the second example gets the User object stored in the first example. Since the getAttribute method returns a value of the Object type, this type must be cast to the User type.

When you work with request attributes, you should realize that the attributes are reset between requests. As a result, if you store a User object as a request attribute and forward that request to a JSP, that User object is only available to that JSP and isn't available later in the session. In chapter 7, you'll learn how to store an object so it's available to any servlet or JSP in the current session.

Two methods available from the HttpServletRequest object

Method	Description
`setAttribute(String name, Object o)`	Stores any object in the request as an attribute and specifies a name for the attribute. Attributes are reset between requests.
`getAttribute(String name)`	Returns the value of the specified attribute as an Object type. If no attribute exists for the specified name, this method returns a null value.

How to set a request attribute

```
User user = new User(firstName, lastName, email);
request.setAttribute("user", user);
```

How to get a request attribute

```
User user = (User) request.getAttribute("user");
```

How to set a request attribute for a primitive type

```
int id = 1;
request.setAttribute("id", new Integer(id));
```

How to get a request attribute for a primitive type

```
int id = (Integer) request.getAttribute("id");
```

Description

- These methods are most often used in conjunction with a RequestDispatcher object that's used to forward a request as described in the next figure.

Figure 5-8 How to get and set request attributes

How to forward requests

Figure 5-9 shows how to *forward* a request from a servlet to an HTML page, a JSP, or another servlet. To do that, you begin by calling the getServletContext method from the HttpServlet class to return a ServletContext object. Then, you call the getRequestDispatcher method of the ServletContext object to return a RequestDispatcher object. Within this method, you must code a URL that starts with a slash so it is relative to the root directory of the current web application. Then, you use the forward method to forward the request and response objects to the HTML page, JSP, or servlet specified by the URL.

In the examples in this figure, the first statement sets the string for the URL. Then, the second statement gets the ServletContext object, gets the RequestDispatcher object for the URL, and forwards the request. In these examples, the second statement uses method chaining as described earlier in this chapter.

How to redirect responses

This figure also shows how to *redirect* a response. To do that, you use the sendRedirect method of the response object. You typically use this technique when you want to transfer control to a URL outside of your application. To use this method, you often supply an absolute URL. However, you can also supply a relative URL because the servlet engine can convert it to an absolute URL. If you begin the pathname with a slash, the servlet engine interprets the path as relative to the root directory for the servlet engine.

When you call the sendRedirect method, the server sends an absolute URL to the browser. Then, the browser sends a request for that URL. Since this processing occurs on the client side rather than the server side, this isn't as efficient as forwarding a request as shown earlier in this figure. In addition, the sendRedirect method doesn't transfer the request and response objects. As a result, you should only use the sendRedirect method when you want to redirect to a URL that's available from another web application.

A method of the ServletContext object for working with paths

Method	Description
`getRequestDispatcher(String path)`	Returns a RequestDispatcher object for the specified path.

A method of the RequestDispatcher object

Method	Description
`forward(ServletRequest request, ServletResponse response)`	Forwards the request and response objects to another resource on the server (usually a JSP or servlet).

How to forward the request to an HTML page

```
String url = "/index.html";
getServletContext().getRequestDispatcher(url)
    .forward(request, response);
```

How to forward the request to a JSP

```
String url = "/thanks.jsp";
getServletContext().getRequestDispatcher(url)
    .forward(request, response);
```

How to forward the request to a servlet

```
String url = "/cart/displayInvoice";
getServletContext().getRequestDispatcher(url)
     .forward(request, response);
```

A method of the HttpServletResponse class

Method	Description
`sendRedirect(String path)`	Sends a temporary redirect response to the client using the specified redirect location URL.

How to redirect a response relative to the current directory

```
response.sendRedirect("email");
```

How to redirect a response relative to the servlet engine

```
response.sendRedirect("/musicStore/email/");
```

How to redirect a response to a different web server

```
response.sendRedirect("http://www.murach.com/email/");
```

Figure 5-9 How to forward requests and redirect responses

How to validate data

When a user enters data into an application, the application often needs to check the data to make sure that the data is valid. This is referred to as data validation. Then, if the user enters data that isn't valid, the application can display an error message and give the user another chance to enter the data.

How to validate data on the client

In a typical web application, the data that's entered by the user is validated by JavaScript in the browser. That way, some invalid entries are caught and fixed before they're sent to the server for processing. That makes the application more responsive to the user and reduces the workload on the server. To learn how to perform data validation on the client, we recommend that you refer to *Murach's JavaScript and jQuery*.

How to validate data on the server

Whether or not the user data is validated on the client, though, it is always validated on the server. One reason for that is that JavaScript on the client can't always do all of the validation that's required. For instance, checking whether an email address is already in the database can only be done on the server. Another reason is that client validation doesn't work if JavaScript is disabled in a user's browser.

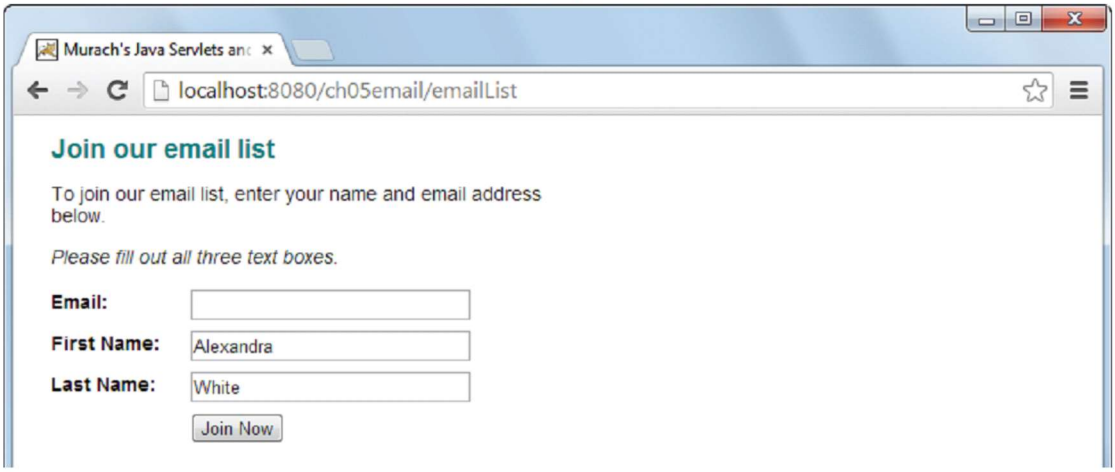
To illustrate data validation on the server, figure 5-10 presents an enhanced version of the Email List application. To start, the code for the first page of the application is now stored in a JSP file, not an HTML file. This allows the first page to use EL to display a validation message like the one shown in this figure, and it allows the first page to use EL to retain any valid values that the user has entered.

When a user requests this JSP for the first time, the page doesn't display a validation message, and it does display empty strings in the three text boxes of the form. Then, if the user submits the data with invalid strings in any of its text boxes, this JSP displays a validation message in italics before the form, and it displays any values that the user has already entered within the text boxes. That way, the user doesn't have to re-enter any valid values that he or she has already entered.

When the user clicks the Join Now button, this page sends the values in the text boxes to the servlet. Then, if a value is still missing, this servlet forwards the request to this JSP again. In other words, this request and response cycle continues until the user enters valid data or ends the application.

In chapter 4, you learned how to use HTML5 to do some rudimentary validation on the client. For example, you learned how to use the required attribute to require the user to enter a value for a text box. In this chapter, the Email List application doesn't include that client-side validation. This allows the developer to test the server-side validation. However, once the developer finishes testing the server-side validation, he or she should restore the client-side validation.

The JSP that's displayed when an entry isn't made



The code for the JSP (index.jsp)

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Murach's Java Servlets and JSP</title>
    <link rel="stylesheet" href="styles/main.css" type="text/css"/>
</head>
<body>
    <h1>Join our email list</h1>
    <p>To join our email list, enter your name and
       email address below.</p>
    <p><i>${message}</i></p>
    <form action="emailList" method="post">
        <input type="hidden" name="action" value="add">
        <label class="pad_top">Email:</label>
        <input type="email" name="email" value="${user.email}"><br>
        <label class="pad_top">First Name:</label>
        <input type="text" name="firstName" value="${user.firstName}"><br>
        <label class="pad_top">Last Name:</label>
        <input type="text" name="lastName" value="${user.lastName}"><br>
        <label> </label>
        <input type="submit" value="Join Now" class="margin_left">
    </form>
</body>
</html>
```

Figure 5-10 How to validate data on the server (part 1 of 2)

Part 2 of this figure shows the code in the doPost method of the servlet that validates the data on the server. If the user is attempting to add a name to the email list, this code begins by getting three request parameters from the form and storing them in a User object. Then, it checks that the user has entered values for all three parameters. If not, this code sets the variable named message to an appropriate validation message, and it sets the variable named url to the JSP that displays the validation message (index.jsp).

Otherwise, this code sets the variable named message to an empty string so no validation message is displayed, it sets the variable named url to the JSP that displays the data to the user (thanks.jsp), and it uses the UserDB class to write the data in the User object to the database.

After the if/else statement, the servlet stores the User object and the message string in the request object. Then, it forwards the request to the specified URL. If, for example, the user hasn't entered valid data, this code forwards the request to the JSP shown in part 1 of this figure.

If you study this servlet, it should give you a better idea of how a servlet can function as a controller in the MVC pattern. In particular, this shows how a servlet can get data from the request, update the model based on that data, and forward the request to the appropriate part of the view.

Keep in mind, though, that checking to make sure an entry has been made in a field is minimal data validation. In practice, the data validation code would also check that the email entry is a valid email address with an at sign (@), a period, and more.

Since this servlet works with strings, it doesn't need any code that converts the data. But what if you want to make sure that the user enters an integer? To do that, you can attempt to convert the string to an int value within a try/catch statement. Then, if the parsing fails and an exception is thrown, the servlet can supply code in the catch statement that displays an appropriate validation message and forwards the request to an appropriate JSP.

A doPost method that validates the data

```
@Override
protected void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {

    String url = "/index.jsp";

    // get current action
    String action = request.getParameter("action");
    if (action == null) {
        action = "join";  // default action
    }

    // perform action and set URL to appropriate page
    if (action.equals("join")) {
        url = "/index.jsp";    // the "join" page
    }
    else if (action.equals("add")) {
        // get parameters from the request
        String firstName = request.getParameter("firstName");
        String lastName = request.getParameter("lastName");
        String email = request.getParameter("email");

        // store data in User object
        User user = new User(firstName, lastName, email);

        // validate the parameters
        String message;
        if (firstName == null || lastName == null || email == null ||
            firstName.isEmpty() || lastName.isEmpty() || email.isEmpty()) {
            message = "Please fill out all three text boxes.";
            url = "/index.jsp";
        }
        else {
            message = "";
            url = "/thanks.jsp";
            UserDB.insert(user);
        }
        request.setAttribute("user", user);
        request.setAttribute("message", message);
    }
    getServletContext()
            .getRequestDispatcher(url)
            .forward(request, response);
}
```

Figure 5-10 How to validate data on the server (part 2 of 2)

How to work with the web.xml file

In chapter 2, you learned that most applications have one web.xml file that contains information about the configuration of the application. Then, earlier in this chapter, you learned how to use the web.xml file to map servlets to URLs. Now, you'll learn more about working with the web.xml file for an application.

The code in this topic has been tested against the Tomcat server. If you're using a different servlet engine, you may need to modify the code presented in this topic so it works with your servlet engine. In general, though, the code presented here should work on all servlet engines.

A complete web.xml file

Figure 5-11 shows the web.xml file that works with the application presented in this chapter. This web.xml file begins with two tags that define the type of XML document that's being used. For now, you don't need to understand this code. However, these tags must be included at the start of each web.xml file. Fortunately, most IDEs create them automatically. Otherwise, you can copy these tags from another web.xml file.

After this code, the web.xml file contains *XML tags* that define *elements*. For example, the opening and closing web-app tags define the web-app element. Since all other elements are coded within this element, the web-app element is known as the *root element*. Any element coded within this element or a lower-level element is known as a *child element*.

To modify a web.xml file, you should be able to use your IDE. As you're making the modifications, if you want to leave code in the XML file but you don't want the servlet engine to use it, you can use XML *comments* to comment out those portions of code. These comments work the same way HTML comments do.

When you're done modifying the file, you must redeploy the application to Tomcat so the changes take effect. With most IDEs, this happens automatically when you run the application. If not, you can always restart Tomcat to make sure that it reads the changes to the web.xml file.

When you modify a web.xml file, you should make sure to nest the XML elements correctly as shown in this figure. Otherwise, when you start Tomcat or attempt to deploy your application, Tomcat won't be able to read the web.xml file. In that case, Tomcat displays an error message. To solve this problem, you can edit the web.xml file and redeploy the application. Or, you can restart Tomcat to read the file again.

A complete web.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
             http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">

    <context-param>
        <param-name>custServEmail</param-name>
        <param-value>custserv@murach.com</param-value>
    </context-param>

    <servlet>
        <servlet-name>EmailListServlet</servlet-name>
        <servlet-class>murach.email.EmailListServlet</servlet-class>
        <init-param>
            <param-name>relativePathToFile</param-name>
            <param-value>/WEB-INF/EmailList.txt</param-value>
        </init-param>
    </servlet>

    <servlet-mapping>
        <servlet-name>EmailListServlet</servlet-name>
        <url-pattern>/emailList</url-pattern>
    </servlet-mapping>

    <!-- you can comment out these tags when the app is in development -->
    <error-page>
        <error-code>404</error-code>
        <location>/error_404.jsp</location>
    </error-page>
    <error-page>
        <exception-type>java.lang.Throwable</exception-type>
        <location>/error_java.jsp</location>
    </error-page>

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

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

Description

- The web.xml file is stored in the WEB-INF directory for an application. When Tomcat starts, it reads the web.xml file.
- If the elements in the web.xml aren't nested correctly, Tomcat displays an error message when it reads the web.xml file.
- After you modify the web.xml file, you must redeploy the application so the changes take effect. Or, you can restart Tomcat.

Figure 5-11 A complete web.xml file

How to work with initialization parameters

If you want to store some *initialization parameters* for an application in a central location, you can add them to the web.xml file as shown in part 1 of figure 5-12. Then, your servlets can read these parameters as shown in part 2 of this figure.

To define a *context initialization parameter* that's available to all servlets in the web application, you code a context-param element. Then, you code two child elements: the param-name element and the param-value element. To define multiple context parameters, you can code additional context-param elements after the first one. In this figure, the context-param element defines a parameter named custServEmail that has a value that provides an email address.

To define a *servlet initialization parameter* that's available to a specific servlet, you can code an init-param element within a servlet element. This element follows the servlet-name and servlet-class elements that you learned about earlier in this chapter.

Within the init-param element, you must code the param-name and param-value elements to define the name and value of the parameter. To define multiple initialization parameters for a servlet, you can code additional init-param elements after the first one. In this figure, the init-param element defines a parameter named relativePathToFile that has a relative path as its value.

With the servlet 3.0 specification (Tomcat 7.0) and later, you can use the @WebServlet and @InitParam annotations to specify servlet initialization parameters. To do that, you can code the @WebServlet annotation as described earlier in this chapter. However, you use the urlPatterns attribute to specify the URL mapping for the servlet. Then, you use the initParams attribute to specify the initialization parameters for the servlet. To do that, you can code one or more @InitParam annotations. Each of these annotations includes name and value attributes that specify the name and value of the parameter.

XML tags that set initialization parameters in a web.xml file

```
<context-param>
    <param-name>custServEmail</param-name>
    <param-value>custserv@murach.com</param-value>
</context-param>

<servlet>
    <servlet-name>AddToEmailListServlet</servlet-name>
    <servlet-class>email.AddToEmailListServlet</servlet-class>
    <init-param>
        <param-name>relativePathToFile</param-name>
        <param-value>/WEB-INF/EmailList.txt</param-value>
    </init-param>
</servlet>
```

XML elements for working with initialization parameters

Element	Description
`<context-param>`	Defines a parameter that's available to all servlets within an application.
`<servlet>`	Identifies a specific servlet within the application.
`<servlet-name>`	Defines the name for the servlet that's used in the rest of the web.xml file.
`<servlet-class>`	Identifies the servlet by specifying the servlet's package and class name.
`<init-param>`	Defines a name/value pair for an initialization parameter for a servlet.
`<param-name>`	Defines the name of a parameter.
`<param-value>`	Defines the value of a parameter.

An annotation that sets initialization parameters for a servlet

```
@WebServlet(urlPatterns={"/emailList"},
            initParams={@InitParam(name="relativePathToFile",
                                   value="/WEB-INF/EmailList.txt")})
```

Description

- To create an *initialization parameter* that's available to all servlets (called a *context initialization parameter*), you code the param-name and param-value elements within the context-param element.
- To create an initialization parameter that's available to a specific servlet (called a *servlet initialization parameter*), you code the param-name and param-value elements within the init-param element. But first, you must identify the servlet by coding the servlet, servlet-name, and servlet-class elements.

Figure 5-12 How to work with initialization parameters (part 1 of 2)

Part 2 of this figure shows you how to read initialization parameters like those in part 1. To retrieve an initialization parameter that's available to all servlets, you begin by calling the getServletContext method from anywhere in the servlet to get a ServletContext object. Then, you call the getInitParameter method from the ServletContext object.

To retrieve an initialization parameter that's available only to the current servlet, you begin by calling the getServletConfig method from anywhere in the servlet to get a ServletConfig object. Then, you call the getInitParameter method from the ServletConfig object.

Note that the getInitParameter method works the same whether you call it from the ServletContext object or the ServletConfig object. The only difference is that the ServletContext object returns parameters that are available to all servlets while the ServletConfig object returns parameters that are only available to the current servlet.

When you call the getInitParameter method, you must specify the name of the parameter. If the parameter exists, the getInitParameter method returns the value of the parameter as a string. Otherwise, this method returns a null value.

Two methods of the GenericServlet class

Method	Description
`getServletContext()`	Returns a ServletContext object that contains information about the entire web application's context.
`getServletConfig()`	Returns a ServletConfig object that contains information about a single servlet's configuration.

A method of the ServletContext and ServletConfig interfaces

Method	Description
`getInitParameter(String name)`	Returns a String object that contains the value of the specified initialization parameter. If the parameter doesn't exist, this method returns a null value.

Code that gets an initialization parameter that's available to all servlets

```
String custServEmail = this.getServletContext()
                           .getInitParameter("custServEmail");
```

Code that gets an initialization parameter that's available to the current servlet only

```
String relativePath = this.getServletConfig()
                          .getInitParameter("relativePathToFile");
```

Description

- To get an initialization parameter that's available to all servlets, you use the getInitParameter method of the ServletContext object.
- To get an initialization parameter for a specific servlet, you use the getInitParameter method of the ServletConfig object.

Figure 5-12 How to work with initialization parameters (part 2 of 2)

How to implement custom error handling

Figure 5-13 shows how to use the web.xml file to specify custom error pages that apply to the entire application. When you're developing an application, you probably won't want to implement custom error pages. That way, when an error occurs, Tomcat displays an error page that you can use to debug the error. Before you deploy an application, though, you may want to implement custom error pages that present errors in a user-friendly way that's consistent with the rest of your application.

To specify a custom error page for a specific *HTTP status code*, you begin by coding an error-page element. Within this element, you code two child elements: the error-code element and the location element. The error-code element specifies the HTTP status code for the error. The location element specifies the location of the custom error page.

The first example in this figure shows how to code the error-page element for the HTTP 404 status code. Here, the error-code element specifies the number for the HTTP status code. Then, the location element specifies a URL that points to an error page named error_404.jsp that's stored in the application's root directory. Note that this element must begin with a slash.

The second example shows some of the code for a custom error page for the 404 status code. This page is a JSP that displays a user-friendly message that describes the HTTP 404 status code.

The 404 status code indicates that the server wasn't able to find a file at the requested URL. As you gain more experience with web programming, you'll become familiar with other HTTP status codes. Also, some of the more common ones are summarized in chapter 18.

To specify a custom error page that's displayed when an uncaught exception is thrown, you begin by coding an error-page element. Within this element, you code two child elements: the exception-type element and the location element. The exception-type element specifies the type of exception by identifying the package name and the class name for the exception. The location element specifies the location of the custom error page.

The third example shows how to code an error-page element that handles all Java exceptions. Here, the exception-type element specifies the Throwable class in the java.lang package. Since all exceptions inherit this class, this causes a custom error page to be displayed for all uncaught exceptions. However, if you want to display different error pages for different types of exceptions, you can code multiple error-page elements. For example, you can display one error page for exceptions of the NullPointerException type and another error page for exceptions of the ServletException type.

The fourth example shows some of the code for a custom error page that handles all Java exceptions. This error page uses EL to display the name of the exception's class and its message. For more information about EL, see chapter 8.

XML elements for working with error handling

Element	Description
`<error-page>`	Specifies an HTML page or JSP that's displayed when the application encounters an uncaught exception or a certain type of HTTP status code.
`<error-code>`	Specifies the number of a valid HTTP status code.
`<exception-type>`	Uses the fully qualified class name to specify a Java exception.
`<location>`	Specifies the location of the HTML page or JSP that's displayed.

The XML tags that provide error-handling for an HTTP 404 status code

```
<error-page>
    <error-code>404</error-code>
    <location>/error_404.jsp</location>
</error-page>
```

The code for a file named error_404.jsp

```
<h1>404 Error</h1>
<p>The server was not able to find the file you requested.</p>
<p>To continue, click the Back button.</p>
```

The XML tags that provide error-handling for all Java exceptions

```
<error-page>
    <exception-type>java.lang.Throwable</exception-type>
    <location>/error_java.jsp</location>
</error-page>
```

The code for a file named error_java.jsp

```
<h1>Java Error</h1>
<p>Sorry, Java has thrown an exception.</p>
<p>To continue, click the Back button.</p>

<h2>Details</h2>
<p>Type: {pageContext.exception["class"]}</p>
<p>Message: {pageContext.exception.message}</p>
```

Figure 5-13 How to implement custom error handling (part 1 of 2)

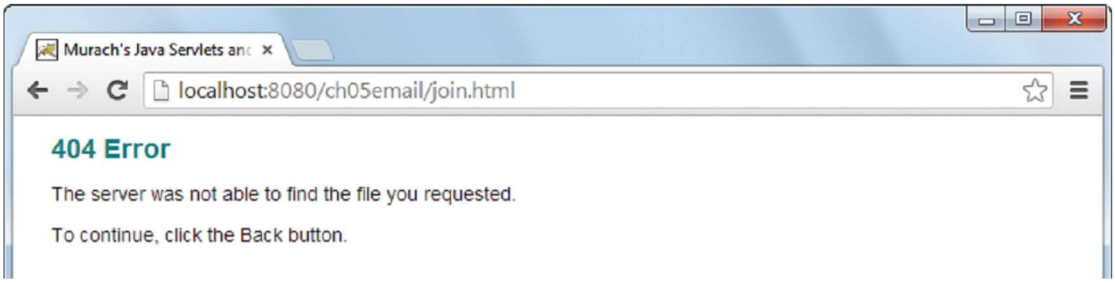
When you code a custom error page, you can use an HTML document or a JSP. If you use a JSP, you can use the exception object to customize the error page. In addition, you can use any other techniques that are available to a JSP.

Part 2 of figure 5-13 shows how the error pages appear when displayed in a browser. Here, the first screen shows the error page that's displayed for an HTTP 404 status code. By default, however, Microsoft Internet Explorer uses its own error pages for HTTP status codes. So if you want to use this browser to view a custom page, you must select the Tools→Internet Options command and use the Advanced tab of the dialog box to deselect the "Show friendly HTTP error messages" option. That's why I used Chrome to display this error page.

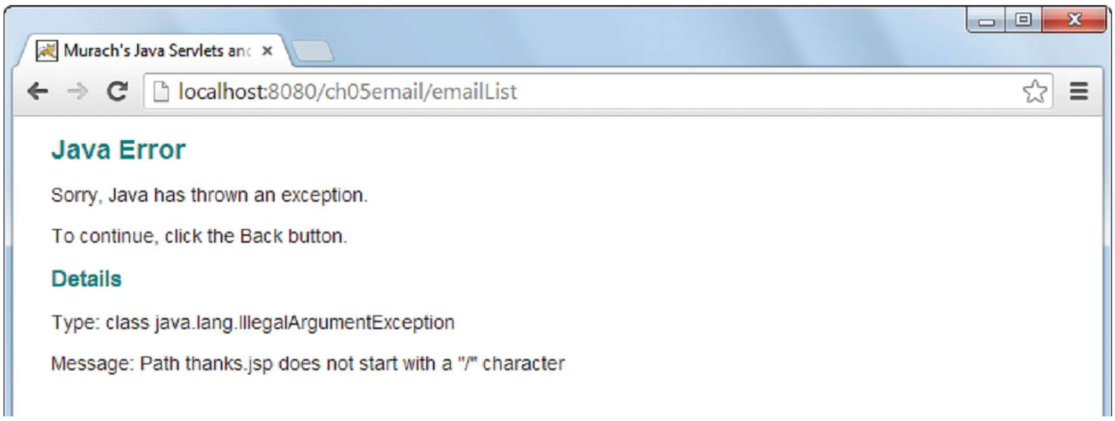
The second screen shows the error page that's displayed when a servlet throws an exception. Here, the details at the bottom of the page show that one of the paths used by the application threw an IllegalArgumentException. In addition, the message indicates that exception was thrown because a front slash wasn't coded at the start of the path for the thanks.jsp file.

In this case, this is all the information you need to debug the error. However, this error page doesn't give as much information as the error pages that are provided by Tomcat. As a result, it's easier to debug errors when you're using Tomcat's error pages. That's why you should only use an error page like this when you're ready to put your web application into a production environment. Until then, you can comment out the error-page element in the web.xml file.

The JSP page for the 404 error



The JSP page that's displayed when a Java exception is thrown



Description

- In the web.xml file, you can use the error-page element to specify the error pages that should be displayed when the application encounters (1) specific HTTP status codes or (2) uncaught exceptions.
- By default, Internet Explorer uses its own error pages for HTTP status codes. As a result, if you want to use this browser to view a custom page, you must select the Tools→Internet Options command and use the Advanced tab of the dialog box to deselect the "Show friendly HTTP error messages" option.

Figure 5-13 How to implement custom error handling (part 2 of 2)

More skills for working with servlets

Now that you have a basic understanding of how to code a servlet, you're ready to learn some other background concepts for working with servlets. Understanding these concepts can help you avoid some common pitfalls. First, it's generally considered a bad practice to override a servlet's service method. Second, it's generally considered a bad practice to code instance variables for a servlet.

How the methods of a servlet work

Figure 5-14 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 initialization code that's required as shown in the next figure.

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.

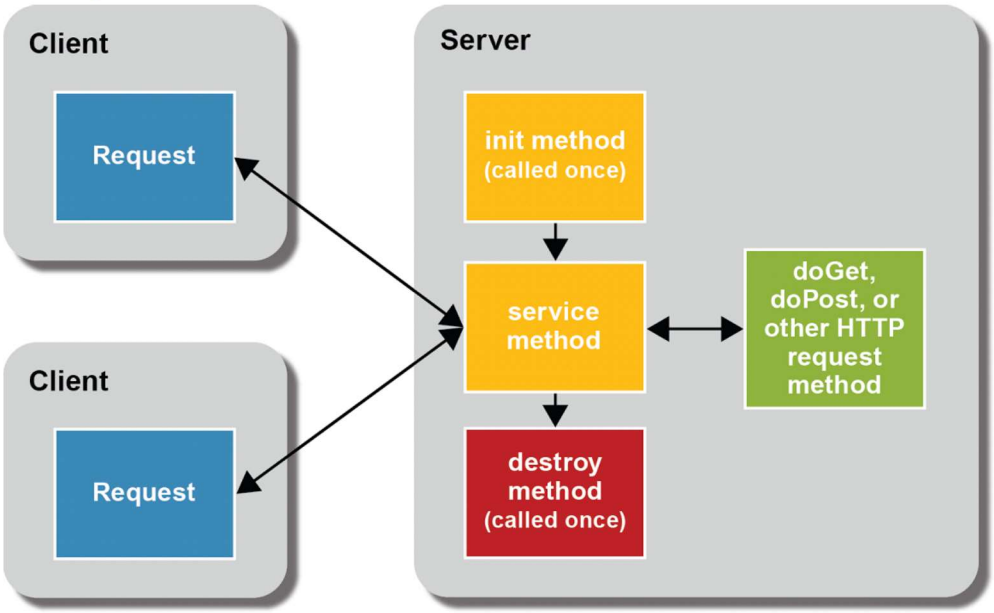
When you code servlets, you shouldn't override the service method. Instead, you should override the appropriate doGet or doPost methods. To handle a request that uses the GET method, for example, you can override the doGet method. If, on the other hand, you want to handle a request that uses the POST method, you can override the doPost method. To use the same code to handle both types of requests, you can override both of them and have one call the other as shown in figure 5-1.

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, you can override this method. However, the destroy method can't 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()
```

The lifecycle of a servlet



Description

- 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 shut down.

Notes

- All the methods shown above are located in the abstract HttpServlet class. As a result, you can override these methods in your own servlets.
- It's generally considered a bad practice to override the service method. Instead, you should override a method like doGet or doPost to handle a specific type of HTTP request.

Figure 5-14 How the methods of a servlet work

Why you shouldn't use instance variables in servlets

Figure 5-15 shows an example of how instance variables were sometimes used with servlets. But using instance variables is not an acceptable practice.

The problem with instance variables is that they aren't thread-safe and they can lead to lost updates and other serious problems. Worse, there's no easy way to make instance variables thread-safe. As a result, you should never code instance variables for a servlet. As you progress through this book, though, you'll learn several thread-safe techniques for working with global variables so you won't have to use instance variables.

Code that adds an instance variable to the EmailServlet class

```
package murach.email;

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

public class EmailListServlet extends HttpServlet
{
    // declare an instance variable for the page
    private int globalCount;  // instance variables are not thread-safe

    @Override
    public void init() throws ServletException
    {
        globalCount = 0;      // initialize the instance variable
    }

    @Override
    protected void doPost(
        HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException
    {
        // update global count variable
        globalCount++;        // this is not thread-safe

        // the rest of the code goes here
    }
}
```

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.
- Instance variables are not *thread-safe*. In other words, two threads may conflict when they try to read, modify, and update the same instance variable at the same time, which can result in lost updates or other problems.
- That's why you should never use instance variables in servlets.

Figure 5-15 Why you shouldn't use instance variables in servlets

How to work with servlet errors

When you develop servlets, you will inevitably encounter errors. That's why this topic gives you some ideas on how to solve common problems and how to print debugging data to the console or to a log file.

How to solve common servlet problems

Figure 5-16 lists four common problems that can occur when you're working with servlets. Then, it lists some possible solutions for each of these problems. Unfortunately, the solutions to these problems vary depending on the IDE you're using. As a result, the solutions presented here are only described in general terms.

If your servlet won't compile, the error message that's displayed 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 make the API available to your application. To do that, you can usually use the IDE to add the appropriate library or JAR file to the application.

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 correctly mapped to the servlet. A common mistake, for example, is to change the name of the servlet and forget to update the mapping for the servlet in the web.xml file.

If you make changes to a servlet and the changes don't show up when you run the servlet, it may be because the servlet engine hasn't reloaded the modified class. In this case, you can enable servlet reloading. That way, the servlet engine automatically detects changes to servlets and reloads them. Or, you can redeploy the application, and many IDEs do this automatically when you run the web application. Finally, if all else fails, you can shut down Tomcat and restart it so Tomcat has to reload all of the applications that are running on it.

If the HTML page that's returned by a servlet doesn't look right when it's rendered by the browser, the servlet is probably sending bad HTML to the browser. To troubleshoot this problem, you can use your browser to view the source HTML that's sent to it. The procedure for doing this varies from browser to browser, but you can typically find a command in the menu system that allows you to view the HTML for the page. Then, you can identify the problem and modify the servlet or JSP 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.
The servlet won't run	Make sure the web server is running. Make sure you're using the correct URL.
Changes to the servlet aren't showing up	Make sure servlet reloading is on. Redeploy the application. Restart the server so it reloads all applications.
The page doesn't display correctly	Use your browser to view the HTML code for the page. Then, you can go through the HTML code to identify the problem, and you can fix the problem in the servlet.

Note

- Chapter 3 shows how to make JAR files available to the NetBeans IDE. You can use a similar technique to make a JAR file available to most other modern IDEs.

Figure 5-16 How to solve common servlet problems

How to print debugging data to the console

You can print debugging messages to the console for the servlet engine as shown in figure 5-17. To do that, you can use the println method of the System.out and System.err objects. You can use these messages to help track the methods that are executed or to view the value of a variable.

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 are easier to understand. This also makes it easier to find and remove the println statements once the error is debugged.

When you use println statements to print debugging data to the console, this data may be printed to different locations depending on your development environment. If, for example, you're using the NetBeans IDE, the data is printed to the Tomcat tab of the Output window. However, if you're using Tomcat in a stand-alone environment, the data is printed to a Tomcat console.

Code that prints debugging data to the console

```
public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
                   throws IOException, ServletException
{
    // code

    String email = request.getParameter("email");
    System.out.println("EmailListServlet email: " + email);

    // more code
}
```

The output that's printed to the Tomcat console

```
EmailListServlet email: jsmith@gmail.com
```

Description

- When you're testing an application on your local system, you can 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 servlet name and the variable name so the messages are easy to interpret.
- When you use NetBeans, it displays a tab for the Tomcat console within its Output window.

Figure 5-17 How to print debugging data to the console

How to print debugging data to a log file

If you want to keep a permanent history of some key debugging data, you can print debugging data to a *log file* as shown in figure 5-18. 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 these methods work.

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. However, 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 first example in this figure uses the first log method to write the value for the emailAddress variable to the log file. Here, Tomcat automatically writes some additional information before the message. First, it writes a timestamp that shows the date and time that this data was written to the log file. Then, it writes the word INFO followed by the name of the servlet class. Note, however, that this information may vary from one servlet engine to another.

The second example in this figure uses the second log method to write a message and a stack trace for an IOException. In this case, Tomcat ignores the message and doesn't write it to the log file. Instead, Tomcat writes some other information to the log file. First, it writes a timestamp that shows the date and time that this stack trace was written to the log file. Then, it writes the word WARNING followed by the name of the method that threw the exception and the name of the servlet class. Finally, it writes the stack trace. Note that the stack trace indicates that the exception was thrown by the 11th line of the UserIO class. As a result, this helps you locate the code that's causing the bug.

Tomcat stores its 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. To view a log file, you can navigate to this directory and open the log file with a text editor. For example, to view the data that's written by the examples in this figure, I opened the log file that's specified at the bottom of this figure. This log file contains the data for any log methods that were executed on the localhost server for June 29th, 2014.

When you use the NetBeans IDE with Tomcat, the server's log file is shown in the Tomcat Log tab of the Output window. As a result, you can view the current log file by displaying that tab.

Two methods of the HttpServlet class used to log errors

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

Servlet code that prints the value of a variable to a log file

```
log("email=" + email);
```

The data that's written to the log file

```
Jun 29, 2014 6:26:04 PM org.apache.catalina.core.ApplicationContext log
INFO: EmailListServlet: email=jsmith@gmail.com
```

Servlet code that prints a stack trace to a log file

```
try {
    UserIO.add(user, path);
}
catch(IOException e) {
    log("An IOException occurred.", e);
}
```

The data that's written to the log file

```
Jun 29, 2014 6:30:38 PM org.apache.catalina.core.StandardWrapperValve invoke
WARNING: Servlet.service() for servlet EmailListServlet threw exception
java.io.FileNotFoundException: C:\murach\servlet_and_jsp\netbeans\ch05email\
build\web\WEB-INF\EmailList.txt (Access is denied)
        at java.io.FileOutputStream.openAppend(Native Method)
        at java.io.FileOutputStream.<init>(FileOutputStream.java:177)
        at java.io.FileWriter.<init>(FileWriter.java:90)
        at murach.data.UserIO.add(UserIO.java:11)
        at murach.email.EmailListServlet.doPost(EmailListServlet.java:38)
...
...
```

The location of a typical Tomcat log file

```
C:\tomcat\logs\localhost.2014-06-29.log
```

Description

- You can use the log methods of the HttpServlet class to write debugging data to a web server's *log file*.
- A *stack trace* is the chain of method calls for any statement that calls a method.
- The data that's written by the log methods varies from one server to another. In fact, Tomcat doesn't include the specified message when you use the second method above. The name and location of the log files may also vary from one server to another.
- When you use NetBeans, it displays a tab for the server log within its Output window.

Figure 5-18 How to print debugging data to a log file

Perspective

The goal of this chapter has been to teach you the basics of coding servlets that work within the MVC pattern. Now, you should be able to develop servlets of your own that act as the controllers of an application. Then, you should be able to forward the HTTP request and response to a JSP that sends the appropriate HTML back to the browser. In the next chapter, you'll learn the details for coding a JSP.

Before you go on to the next chapter, though, you should know that some of the skills described in this chapter also apply to JSPs. For example, you can use the log method within a JSP to write data to a log file. However, if you use the MVC pattern, you typically won't need to add any log statements to your JSPs.

Summary

- A *servlet* is a Java class that extends the HttpServlet class and runs within a servlet container such as Tomcat.
- When you write servlets, you typically 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.
- To request a servlet, you request a URL that has been mapped to the servlet. This mapping is specified in the web.xml file by the servlet and servlet-mapping elements. Or, with the servlet 3.0 specification (Tomcat 7.0) or later, you can use the @WebServlet annotation to map a servlet to a URL.
- When you use the HTTP GET method to pass parameters to a JSP, the browser displays the parameters in its URL. When you use the HTTP POST method, the browser doesn't display these parameters. If executing a request multiple times may cause problems, you should use the POST method.
- Within a servlet's doGet or doPost method, you can use the getParameter and getParameters methods of the request object to get the parameter values from the request.
- You can use the getRealPath method of the ServletContext object to get a relative path to refer to a file that's used by your web application.
- The web.xml file consists of *XML tags* that define *XML elements*. The *root element* for this file is the web-app element. When one element is coded within another element, it can be called a *child element*.

- You can use the web.xml file or @WebServlet annotations to provide *initialization parameters* that apply to the entire web application or to specific servlets.
- You can use the web.xml file to provide custom error pages for specific Java exceptions or for HTTP errors represented by specific *HTTP status codes*.
- You can use JavaScript or jQuery to perform *data validation* on the client. You can use a servlet to perform data validation on the server. It's considered a best practice to perform data validation on both the client and the server.
- It's generally considered a bad practice to override the service method of a servlet.
- Since instance variables in a servlet are not thread-safe and can result in serious problems, you should never use instance variables in a servlet.
- To print debugging data to the server console, you can use the println method of the System.out or System.err object. Or, you can use the log methods of the HttpServlet class to write debugging data to a *log file*.

Exercise 5-1 Modify the servlet for the Email List application

In this exercise, you'll modify the servlet that's used by the email application that's presented in this chapter.

1. Open the ch05_ex1_email project in the ex_starts directory, and review its files. This should include an EmailListServlet class. Note that the EmailListServlet class validates the data entered by the user and that this class does not contain a doGet method.
2. Run the application. Then, enter some valid values on the first web page and click on the Join Now button to run the EmailListServlet class. This should use the HTTP POST method to request the /emailList URL, and the browser should not show any parameters in its URL.
3. Click on the Return button. This should use the HTTP POST method to request the /emailList URL, it should display the first page of the application, and the browser should not show any parameters in its URL.
4. Click the Join Now button without entering any values. This should display the same page again, but with a validation message that indicates that all three values are required.
5. Append two parameters to the /emailList URL that's displayed by your browser like this:

   ```
   /emailList?action=add&email=jsmith@gmail.com
   ```

   This uses the HTTP GET method. Since the EmailListServlet doesn't have a doGet method, this should display an error page that indicates that the HTTP GET method isn't supported by the URL.

6. Open the code for the servlet and add a doGet method that calls the doPost method of the servlet. Then, run the application again and enter the /emailList URL with the same parameters. It should display the first page of the web application with a validation message.

   This tests whether your development environment automatically reloads servlets after they have been changed. If the change you made to the servlet isn't working, you may need to manually reload the servlet class. One way to do that is to stop and restart Tomcat.

7. Open the code for the servlet and add a statement that prints a debugging message to the Tomcat console. This message should show the value of the action parameter that's passed to the servlet.
8. Run the application and click the Join Now button to display the message in the console. If you're using NetBeans, it should display a tab for the Tomcat console within the Output window. If it doesn't, you can usually display this tab by using the Services tab to restart the Tomcat server.
9. Repeat the previous two steps, but use a log file this time. If you're using NetBeans, it should display a tab for the Tomcat log file within its Output window.

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 5-1, but the details are repeated here.

1. Open the ch05_ex2_email project that's in the ex_starts directory.
2. Add a servlet named TestServlet to the murach.email package and map that servlet to the /test URL.
3. Modify the code for the servlet so it implements the doPost method and the doGet method, but not the service method.
4. Modify the doGet method so it displays a message that says "TestServlet Get".
5. Modify the doPost method so it displays a message that says "TestServlet Post".
6. Run the project and enter the /test URL into the browser to run the test servlet. This should show that the test servlet works for the HTTP GET method.
7. Open the index.jsp file and modify it so it calls the /test URL instead of the /emailList URL.
8. Run the project and click the Join Now button to run the test servlet. This should show that the test servlet works for the HTTP POST method.
9. If the project uses the web.xml file to map the servlet, delete the mapping from the web.xml file and use the @WebServlet annotation to map the servlet instead.
10. Run the project and test the servlet to make sure that the new mapping works.