IN THIS CHAPTER

- Looking at servlets
- Downloading, installing, and configuring Tomcat
- Creating simple servlets
- Working with forms to get data from the user

Chapter 1
Creating Servlets

Servlets are among the most popular ways to develop web applications today. Many of the best-known websites are powered by servlets. In this chapter, I give you just the basics: what a servlet is, how to set up your computer so that you can code and test servlets, and how to create a simple servlet. The next two chapters build on this topic, presenting additional web programming techniques.

Understanding Servlets

Before you can understand what a servlet is and how it works, you need to understand the basics of how web servers work. Web servers use a networking protocol called HTTP to send web pages to users. (*HTTP* stands for *Hypertext Transfer Protocol,* but that won't be on the test.) With HTTP, a client computer uses a uniform resource locator, or URL, to request a document that's located on the server computer. HTTP uses a *request/response model,* which means that client computers (web users) send request messages to HTTP servers, which in turn send response messages back to the clients.

A basic HTTP interaction works something like this:

1. Using a web-browser program running on a client computer, you specify the URL of a file that you want to access.

   In some cases, you actually type the URL of the address, but most of the time you click a link that specifies the URL.

2. Your web browser sends an HTTP request message to the server computer indicated by the URL.

   The request includes the name of the file that you want to retrieve.

3. The server computer receives the file request, retrieves the requested file, and sends the file back to you in the form of an HTTP response message.

4. The web browser receives the file, interprets the contents of the file (usually a mix of HTML, JavaScript, and CSS), and displays the result onscreen.

The most important thing to note about normal web interactions is that they're static. By *static*, I mean that the content of the file sent to the user is always the same. If the user requests the same file 20 times in a row, the same page displays 20 times.

By contrast, a servlet provides a way for the content to be dynamic. A *servlet* is simply a Java program that extends the `javax.servlet.Servlet` class. The `Servlet` class enables the program to run on a web server in response to a user request, and output from the servlet is sent back to the web user as an HTML page.

When you use servlets, Steps 1, 2, and 4 of the preceding procedure are the same; the fateful third step is what sets servlets apart. If the URL specified by the user refers to a servlet rather than a file, Step 3 goes more like this:

3. The server receives the servlet request, locates the Java servlet indicated by the request and passes the request to the servlet. The servlet then generates an appropriate response, sends it back to the client via an HTTP response message, and kicks back until another request is delivered to the servlet.

In other words, instead of sending the contents of a file, the server sends the output generated by the servlet. Typically, the servlet program generates a mixture of HTML, JavaScript, and CSS that's displayed by the browser.

Servlets are designed to get their work done quickly and then end. The first time a servlet is run, it processes one request from a browser, generates one page and sends it back to the browser, and then patiently waits for another request. A single servlet can handle thousands or millions of requests, but each request is treated as an independent process: The request is received and processed, and a result is sent back.

If the server decides that a servlet hasn't seen any requests for a while, it may shut down the servlet. In that case, the servlet will be started up again automatically the next time a request for it is received by the container.

Using Tomcat

Unfortunately, you can't run servlet programs on any old computer. First, you have to install a special program called a *servlet container* to turn your computer into a server that's capable of running servlets. The best-known servlet container is Apache Tomcat, which is available free from the Apache Software Foundation at `http://tomcat.apache.org`. For this chapter, I used Tomcat version 10.

Tomcat can also work as a basic web server. In actual production environments, Tomcat is usually used in combination with a specialized web server, such as Apache's HTTP Server.

Installing Tomcat

Installing Tomcat is a simple as installing any other piece of software: You simply download and run an installer. You can find the installer at `https://tomcat.apache.org/download-10.cgi`. This page lists several downloads. The one you want is found under Binary Distributions/Core and is named 32-bit/64-bit Windows Service Installer.

The installer runs like any other Windows installer. You can accept all the defaults if you want, but you may want to change a few defaults along the way — specifically, the following:

- On the Change Components page, I recommend you switch from Normal to Full installation. This will give you all Tomcat components, including sample programs.
- On the Configuration page, provide a username and password for the Administrator logon.
- On the Java Virtual Machine page, you can change the path to the Java JRE if you want. By default, it will pick the Java 8 JRE if that's installed on your system. If you have a more current version, you can select it instead. For example, you can enter **C:\Program Files\Java\jdk-14.0.1** to select JDK 14.
- On the final page of the installation wizard, you can select Run Apache Tomcat to automatically start the Tomcat service. That way, you won't have to manually start it.

When the installation completes, an icon appears in the System Tray to allow you to control Tomcat. This Tray icon offers commands that let you configure various Tomcat options, start or stop the Tomcat service, and create a Thread Dump, which can be useful for debugging purposes.

Testing Tomcat

To find out whether you installed Tomcat correctly, you can try running the test servlets that are automatically installed when you install Tomcat. Open a web-browser window, and type this address:

```
http://localhost:8080
```

The page shown in Figure 1-1 appears.

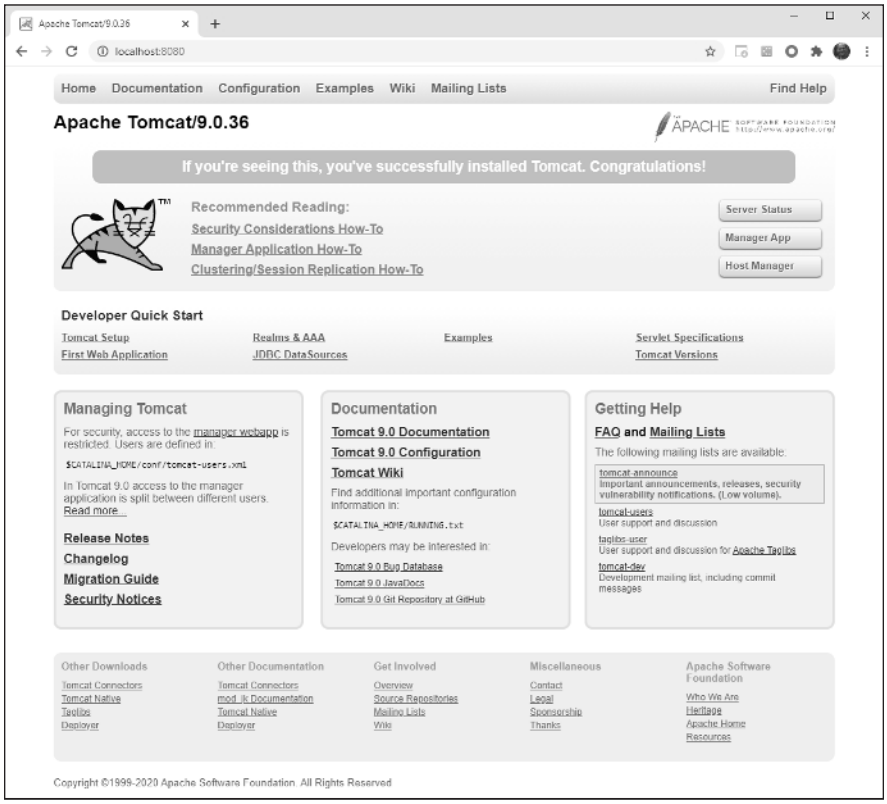

FIGURE 1-1: Tomcat's home page.

You can view interesting information about Tomcat by clicking the Server Status button, located near the upper right of the home page. Figure 1-2 shows the Server Status page. Here, you can see plenty of information about the Tomcat server. One item you may want to note is the JVM version. In Figure 1-2, you can verify that JVM 14.0.1 is active.

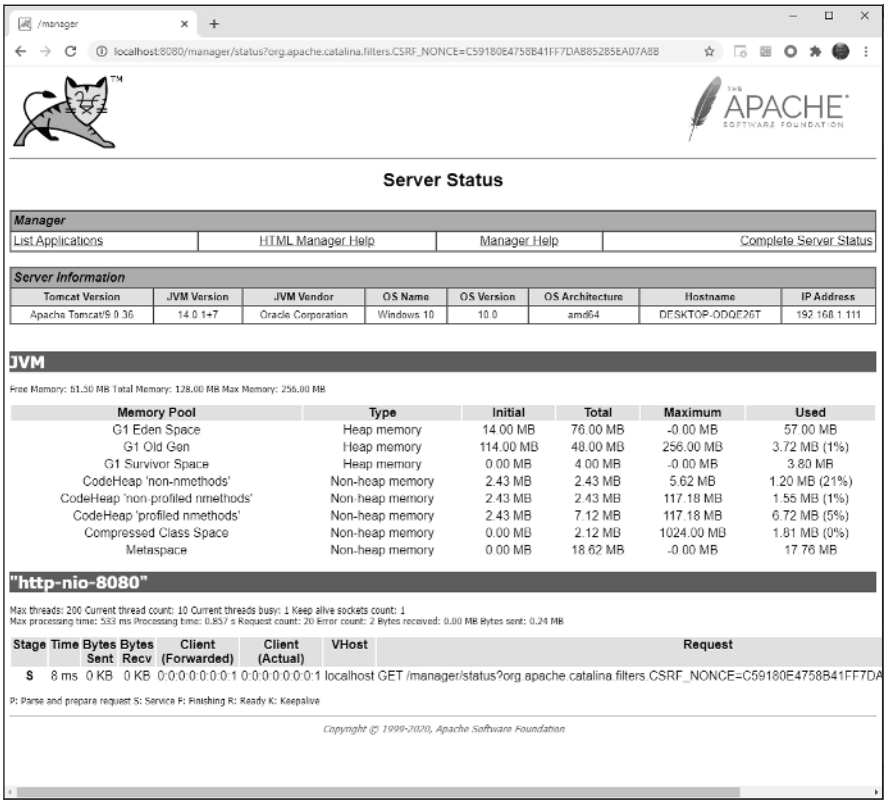

FIGURE 1-2: Tomcat's server status page.

Creating a Simple Servlet

Okay, enough of the configuration stuff; now you can start writing some code. The following sections go over the basics of creating a simple servlet named `HelloWorld`, which simply displays the text "Hello, World!" in the browser window. This servlet will be housed in a package named `com.lowewriter.helloworld`.

Creating the folder structure for a servlet

Before you start writing code, take a few minutes to set up a folder structure to hold all the bits and pieces of your application. Start by creating a root folder. You can put this folder anywhere you want, and you can name it anything you want. For this example, I've named the folder `HelloWorldServlet`.

Next, create the following subfolders within that root folder:

- `src`: This folder will hold your `.java` source files.
- `deploy`: This folder will hold the deployment image for the application.
- `web`: This folder can hold any static HTML or JSP pages. You should also create a folder named `WEB-INF` in the web folder as explained next.
- `web\WEB-INF`: This folder holds an important configuration file named `web.xml`. You should also create a folder named `classes` within the `WEB-INF` folder, as described next.
- `web\WEB-INF\classes`: This folder will hold the `.class` files for your application. Note that if your application uses a package, you should create a folder hierarchy that corresponds to your package name. For example, if the package is `com.lowewriter.helloworld`, you'll need to create the corresponding folder `com\lowewriter\helloworld` within the `classes` folder.

The resulting folder structure for the `HelloWorld` servlet looks like this:

```
HelloWorldServlet
    src
    deploy
    web
        WEB-INF
            classes
                com
                    lowewriter
                        helloworld
```

Creating the web.xml file

Every Java servlet application requires a file named `web.xml` in the `web\WEB-INF` folder. This file specifies several critical attributes of the servlet application, including the name of the application, the name of the servlet class, and the URL used to invoke the servlet.

For more information about XML, refer to chapter 5 of the bonus content.

The `web.xml` file for the `HelloWorld` servlet application is shown in Listing 1-1.

LISTING 1-1: The web.xml File for the HelloWorld Servlet

```
<?xml version="1.0" encoding="UTF-8"?>                                    →1
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"                       →2
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                      http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
  version="4.0"
  metadata-complete="true">

    <display-name>Hello World!</display-name>                             →9

    <servlet>                                                             →11
        <servlet-name>helloworld</servlet-name>
        <servlet-class>com.lowewriter.helloworld.HelloWorld</servlet-class>
    </servlet>

    <servlet-mapping>                                                     →16
        <servlet-name>helloworld</servlet-name>
        <url-pattern>/Hello</url-pattern>
    </servlet-mapping>

</web-app>
```

The following paragraphs describe the key elements of this file:

- →1 This `?xml` element is a standard header required at the start of all XML files.
- →2 The `web-app` element provides basic information about the application. This information is the same for all `web.xml` files.
- →9 The `display-name` element provides a descriptive name for the application.
- →11 The `servlet` tag associates a servlet with a class. The servlet tag has two subelements:
  - `servlet-name`: This element provides the name of the servlet. In Listing 1-1, the name is `helloworld`.
  - `servlet-class`: This element provides the fully qualified name of the class that implements the servlet. In Listing 1-1, the name is `com.lowewriter.helloworld.HelloWorld`.

» →16 The servlet mapping element associates a URL with a servlet. It has two subelements:

- `servlet-name`: This element should be the same as the corresponding subelement of the servlet element — in this case, `helloworld`.
- `URL-pattern`: This element indicates the URL that will be used to invoke the servlet. Note that the URL here is relative to the application's root. Thus, `/Hello` means that the user can invoke the `HelloWorld` servlet by specifying *host*`/helloworld/Hello`, where *host* identifies the web host. (See "Running a Servlet," later in this chapter, for more information.)

In this simple example, only one servlet is defined for the application, and that servlet is associated with just one URL. However, more complicated applications can include more than one `servlet` element, and each servlet can have multiple `URL-pattern` elements that provide aliases for the servlet.

Now that you've set up the folder structure and created the `web.xml` file, you're ready to get on to writing code. To create the code for a servlet class, create a `.java` file in the `src` folder. The next few sections show you how to write the Java code for the `HelloWorld` servlet.

Importing the servlet packages

Most servlets need access to at least three packages: `javax.servlet`, `javax.servlet.http`, and `java.io`. As a result, you usually start with these `import` statements:

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

Depending on what other processing your servlet does, you may need additional `import` statements.

Extending the HttpServlet class

To create a servlet, you write a class that extends the `HttpServlet` class. Table 1-1 lists six methods you can override in your servlet class.

Most servlets override at least the `doGet` method. This method is called by the servlet engine when a user requests the servlet by typing its address in the browser's address bar or by clicking a link that leads to the servlet.

TABLE 1-1 The HttpServlet Class

Method	When Called	Signature
doDelete	HTTP DELETE request	public void doDelete(Http ServletRequest request, HttpServletResponse response)throws IOException, ServletException
doGet	HTTP GET request	public void doGet(HttpServletRequest request, HttpServletResponse response) throwsIOException, ServletException
doPost	HTTP POST request	public void doPost(HttpServletRequest request, HttpServletResponse response) throws IOException, ServletException
doPut	HTTP PUT request	public void doPut(HttpServletRequest request, HttpServletResponse response) throws IOException, ServletException
init()	First time servlet is run	public void init() throws ServletException
destroy()	Servlet is destroyed	public void destroy()

Two parameters are passed to the `doGet` method:

- An `HttpServletRequest` object representing the incoming request from the user. You use the `request` parameter primarily to retrieve data entered by the user in form fields. You find out how to do that later in this chapter, in the section "Getting Input from the User."
- An `HttpServletResponse` object representing the response that is sent back to the user. You use the `response` parameter to compose the output that is sent back to the user. You find out how to do that in the next section.

Printing to a web page

One of the main jobs of most servlets is writing HTML output that's sent back to the user's browser. To do that, you first call the `getWriter` method of the `HttpServletResponse` class, which returns a `PrintWriter` object that's connected to the response object. Thus, you can use the familiar `print` and `println` methods to write HTML text.

Here's a `doGet` method for a simple `HelloWorld` servlet:

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

```
{
    PrintWriter out = response.getWriter();
    out.println("Hello, World!");
}
```

Here the `PrintWriter` object returned by `response.getWriter()` is used to send a simple text string back to the browser. If you run this servlet, the browser displays the text `Hello, World!`.

Responding with HTML

In most cases, you don't want to send simple text back to the browser. Instead, you want to send formatted HTML. To do that, you must first tell the response object that the output is in HTML format. You can do that by calling the `setContentType` method, passing the string `"text/html"` as the parameter. Then you can use the `PrintWriter` object to send HTML.

Listing 1-2 shows a basic `HelloWorld` servlet that sends an HTML response.

LISTING 1-2: **The HelloWorld Servlet**

```
package com.lowewriter.helloworld;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloWorld extends HttpServlet
{
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
            throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>HelloWorld</title>");
        out.println("</head>");
        out.println("<body>");
```

```
        out.println("<h1>Hello, World!</h1>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

Here the following HTML is sent to the browser (I added indentation to show the HTML's structure):

```
<html>
  <head>
    <title>HelloWorld</title>
  </head>

<body>
    <h1>Hello, World!</h1>
  </body>
</html>
```

When run, the `HelloWorld` servlet produces the page shown in Figure 1-3.

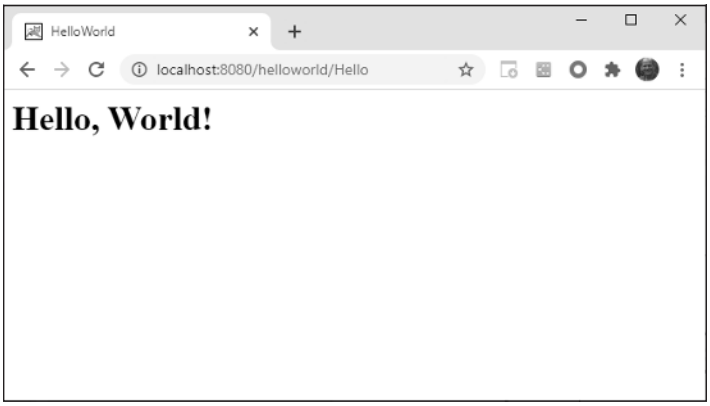

FIGURE 1-3: The `HelloWorld` servlet displayed in a browser.

TIP

Obviously, you need a solid understanding of HTML to write servlets. If HTML is like a foreign language, you need to pick up a good HTML book, such as *HTML, XHTML, & CSS For Dummies*, 7th Edition, by Ed Tittel and Jeff Noble (Wiley), before you go much further.

For your reference, Table 1-2 summarizes all the HTML tags that I use in this book.

TABLE 1-2 Just Enough HTML to Get By

HTML tag	Description
`<html>,</html>`	Marks the start and end of an HTML document.
`<head>,</head>`	Marks the start and end of the head section of an HTML document.
`<title>,</title>`	Marks a title element. The text between the start and end tags is shown in the title bar of the browser window.
`<body>,</body>`	Marks the start and end of the body section of an HTML document. The content of the document is provided between these tags.
`<h1>,</h1>`	Formats the text between these tags as a level-1 heading.
`<h2>,</h2>`	Formats the text between these tags as a level-2 heading.
`<h3>,</h3>`	Formats the text between these tags as a level-3 heading.
`<form action="url", method="method">`	Marks the start of a form. The `action` attribute specifies the name of the page, servlet, or JavaServer Page (JSP) the form is posted to. The `method` attribute can be `GET` or `POST`; it indicates the type of HTTP request sent to the server.
`</form>`	Marks the end of a form.
`<input type="type", name="name">`	Creates an input field. Specify `type="text"` to create a text field or `type="submit"` to create a Submit button. The `name` attribute provides the name you use in the program to retrieve data entered by the user.
` `	Represents a nonbreaking space.

Running a Servlet

So exactly how do you run a servlet? Follow these steps:

1. **Compile the `.java` file to create a `.class` file.**

   Use the `javac` command.

2. **Move the `.class` file into the correct class directory.**

   For example, the directory might be `web\classes\com\lowewriter\helloworld`.

3. **Create a `.war` file for the application.**

   A `.war` file is a special type of `.jar` file that's used to deploy web applications. The `.war` file will be placed in the application's `deploy` folder.

To create the .war file, open a command prompt and navigate to your application's root folder. Then enter this command:

```
jar cvf deploy\helloworld.war –C web .
```

This command creates the file helloworld.war in the deploy folder. The .war file will contain the entire contents of the web folder. Note that the period at end of the command is required.

4. **Copy the .war file into Tomcat's webapps folder.**

   For example, the folder might be C:\Program Files\Apache Software Foundation\Tomcat 10.0\webapps. When you drop the .war file there, Tomcat automatically installs the application. Within a few moments, you see a folder named helloworld appear in the webapps folder.

5. **Open a web browser and enter the URL for the servlet.**

   For the HelloWorld servlet, enter the following URL:

```
http://localhost:8080/helloworld/Hello
```

Improving the HelloWorld Servlet

The HelloWorld servlet, shown in Listing 1-2 earlier in this chapter, isn't very interesting because it always sends the same text. Essentially, it's a static servlet — which pretty much defeats the purpose of using servlets in the first place. You could just as easily have provided a static HTML page.

Listing 1-3 shows the code for a more interesting version called RandomHello. This version uses the random method of the Math class to pick a random number from 1 to 6 and then uses this number to decide which greeting to display.

LISTING 1-3: The RandomHello Servlet

```
package com.lowewriter.randomhello;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
public class HelloServlet extends HttpServlet
{
    public void doGet(HttpServletRequest request,
```

(continued)

LISTING 1-3: ***(continued)***

```
        HttpServletResponse response)
            throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String msg = getGreeting();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>HelloWorld Servlet</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>");
        out.println(msg);
        out.println("</h1>");
        out.println("</body>");
        out.println("</html>");
    }

    private String getGreeting()
    {
        String msg = "";
        int rand = (int)(Math.random() * (6)) + 1;
        switch (rand)
        {
            case 1:
                return "Hello, World!";
            case 2:
                return "Greetings!";
            case 3:
                return "Felicitations!";
            case 4:
                return "Yo, Dude!";
            case 5:
                return "Whasssuuuup?";
            case 6:
                return "Hark!";
        }
        return null;
    }
}
```

Getting Input from the User

If a servlet is called by an HTTP GET or POST request that came from a form, you can call the getParameter method of the request object to get the values entered by the user in each form field. Here's an example:

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

Here the value entered in the form input field named name is retrieved and assigned to the String variable name.

Working with forms

As you can see, retrieving data entered by the user in a servlet is easy. The hard part is creating a form in which the user can enter the data. To do that, you create the form by using a separate HTML file. Listing 1-4 shows an HTML file named InputServlet.html that displays the form shown in Figure 1-4.

LISTING 1-4: The InputServlet.html File

```
<html>
  <head>
    <title>Input Servlet</title>
  </head>

<body>
    <form action="/servlet/InputServlet"
        method="post">
      Enter your name: 
      <input type="text" name="Name">
      <br><br>
      <input type="submit" value="Submit">
    </form>
  </body>
</html>
```

The action attribute in the form tag of this form specifies that /servlet/InputServlet is called when the form is submitted, and the method attribute indicates that the form is submitted via a POST rather than a GET request.

FIGURE 1-4: A simple input form.

The form itself consists of an input text field named `name` and a Submit button. It's nothing fancy — just enough to get some text from the user and send it to a servlet.

Using the InputServlet servlet

Listing 1-5 shows a servlet that can retrieve the data from the form shown in Listing 1-3.

LISTING 1-5: **The InputServlet Servlet**

```
package com.lowewriter.inputservlet;

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

public class InputServlet extends HttpServlet
{
    public void doGet(HttpServletRequest request,
         HttpServletResponse response)
            throws IOException, ServletException
    {
        String name = request.getParameter("Name");
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Input Servlet</title>");
        out.println("</head>");
        out.println("<body>");
```

```
        out.println("<h1>");
        out.println("Hello " + name);
        out.println("</h1>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

As you can see, this servlet really isn't much different from the first `HelloWorld` servlet in Listing 1-2. The biggest difference is that it retrieves the value entered by the user in the `name` field and uses it in the HTML that's sent to the response `PrintWriter` object. If the user enters **Calvin Coolidge** in the `name` input field, for example, the following HTML is generated:

```
<html>
  <head>
    <title>HelloWorld</title>
  </head>

<body>
    <h1>Hello Calvin Coolidge</h1>
  </body>
</html>
```

Thus the message `Hello Calvin Coolidge` is displayed on the page.

Although real-life servlets do a lot more than just parrot back information entered by the user, most of them follow this surprisingly simple structure — with a few variations, of course. Real-world servlets validate input data and display error messages if the user enters incorrect data or omits important data, and most real-world servlets retrieve or update data in files or databases. Even so, the basic structure is pretty much the same.

Using Classes in a Servlet

When you develop servlets, you often want to access other classes that you've created, such as input/output (I/O) classes that retrieve data from files or databases, utility or helper classes that provide common functions such as data validation, and perhaps even classes that represent business objects such as customers or products. If you save all your classes in the same folder, the `javac` compiler will be able to find them when you compile the classes, and the `jar` command will combine them into the `.war` file.

To illustrate a servlet that uses several classes, Figure 1-5 shows the output from a servlet that lists movies read from a text file. This servlet uses three classes:

- `Movie`: A class that represents an individual movie.
- `MovieIO`: A class that has a static public method named `getMovies`. This method returns an `ArrayList` object that contains all the movies read from the file.
- `ListFiles`: The main servlet class. It calls the `MovieIO.getMovies` class to get an `ArrayList` of movies and then displays the movies on the page.

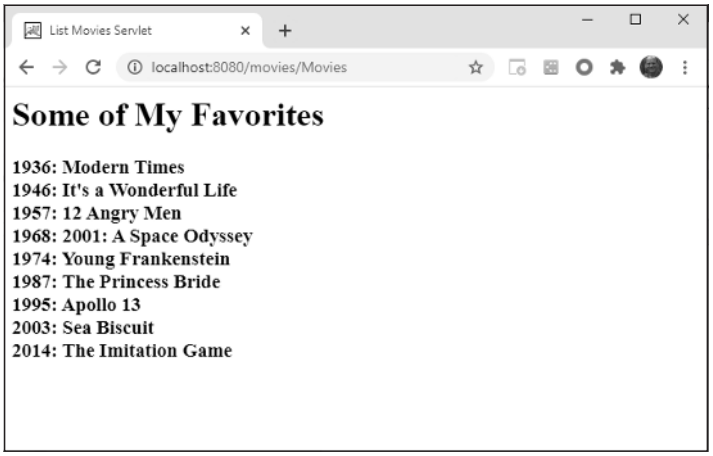

FIGURE 1-5: The ListMovies servlet.

The code for the `Movie` class is shown in Listing 1-6. As you can see, this class doesn't have much: It defines three public fields (`title`, `year`, and `price`) and a constructor that lets you create a new `Movie` object and initialize the three fields. Note that this servlet doesn't use the `price` field.

LISTING 1-6: The Movie Class

```
public class Movie
{
    public String title;
    public int year;
    public double price;
    public Movie(String title, int year,
                 double price)
    {
        this.title = title;
```

```
        this.year = year;
        this.price = price;
    }
}
```

Listing 1-7 shows the `MovieIO` class. This class uses the file I/O features that are presented in chapter 2 of the bonus content to read data from a text file. The text file uses tabs to separate the fields and contains these lines:

```
Modern Times→1936→14.95
It's a Wonderful Life→1946→15.95
12 Angry Men→1957→14.95
2001: A Space Odyssey→1968→19.95
Young Frankenstein→1974→16.95
The Princess Bride→1987→16.95
Apollo 13→1995→19.95
Sea Biscuit→2003→12.95
The Imitation Game→2014→17.95
```

Here the arrows represent tab characters in the file. I'm not going to go over the details of this class here, except to point out that `getMovies` is the only public method in the class, and it's static, so you don't have to create an instance of the `MovieIO` class to use it. For details on how this class works, turn to chapter 2 of the bonus content.

LISTING 1-7: **The MovieIO Class**

```
package com.lowewriter.movie;

import java.io.*;
import java.util.*;

public class MovieIO
{
    public static ArrayList<Movie> getMovies()
    {
        ArrayList<Movie> movies =
            new ArrayList<Movie>();
        BufferedReader in =
            getReader("movies.txt");
        Movie movie = readMovie(in);
        while (movie != null)
        {
            movies.add(movie);
            movie = readMovie(in);
```

(continued)

LISTING 1-7: *(continued)*

```
        }
        return movies;
    }

    private static BufferedReader getReader(
        String name)
    {
        BufferedReader in = null;
        try
        {
            File file = new File(name);
            in = new BufferedReader(
                new FileReader(file) );
        }
        catch (FileNotFoundException e)
        {
            System.out.println(
                "The file doesn't exist.");
            System.exit(0);
        }
        return in;
    }

    private static Movie readMovie(BufferedReader in)
    {
        String title;
        int year;
        double price;
        String line = "";
        String[] data;
        try
        {
            line = in.readLine();
        }
        catch (IOException e)
        {
            System.out.println("I/O Error");
            System.exit(0);
        }
        if (line == null)
            return null;
        else
        {
            data = line.split("\t");
            title = data[0];
```

```
            year = Integer.parseInt(data[1]);
            price = Double.parseDouble(data[2]);
            return new Movie(title, year, price);
        }
    }
}
```

Listing 1-8 shows the code for the `ListMovie` servlet class.

LISTING 1-8: **The ListMovie Servlet Class**

```
package com.lowewriter.movie;

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

public class ListMovies extends HttpServlet
{
   public void doGet(HttpServletRequest request,                          →10
      HttpServletResponse response)
          throws IOException, ServletException
   {
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      String msg = getMovieList();
      out.println("<html>");
      out.println("<head>");
      out.println("<title>List Movies Servlet</title>");
      out.println("</head>");
      out.println("<body>");
      out.println("<h1>Some of My Favorites</h1>");
      out.println("<h3>");
      out.println(msg);
      out.println("</h3>");
      out.println("</body>");
      out.println("</html>");
   }
   public void doPost(HttpServletRequest request,                         →30
      HttpServletResponse response)
          throws IOException, ServletException
```

(continued)

LISTING 1-8: ***(continued)***

```
    {
        doGet(request, response);
    }

    private String getMovieList()                                          →37
    {
        String msg = "";
        ArrayList<Movie> movies = MovieIO.getMovies();
        for (Movie m : movies)
        {
            msg += m.year + ": ";
            msg += m.title + "<br>";
        }
        return msg;
    }
}
```

The following paragraphs describe what its methods do:

- →10 The `doGet` method calls the `getMovieList` method to get a string that contains a list of all the movies, separated by break (`<br>`) tags. Then it uses a series of `out.println` statements to write HTML that displays this list.
- →30 The `doPost` method simply calls the `doGet` method. That way, the servlet works whether it is invoked by a `GET` or a `POST` request.
- →37 The `getMovieList` method calls the `MovieIO.getMovies` method to get an `ArrayList` that contains all the movies read from the file. Then it uses an enhanced `for` loop to retrieve each `Movie` object. Each movie's year and title is added to the `msg` string, separated by `<br>` tags.