Chapter 2:  
A Fast Introduction to Basic Servlet Programming

Topics in This Chapter

- The advantages of servlets over competing technologies
- The basic servlet structure and life cycle
- Servlet initialization parameters
- Access to form data
- HTTP 1.1 request headers, response headers, and status codes
- The servlet equivalent of the standard CGI variables
- Cookies in servlets
- Session tracking
Servlets are Java technology’s answer to Common Gateway Interface (CGI) programming. They are programs that run on a Web server, acting as a middle layer between a request coming from a Web browser or other HTTP client and databases or applications on the HTTP server. Their job is to perform the following tasks, as illustrated in Figure 2–1.

**Figure 2–1** The role of Web middleware.

1. **Read the explicit data sent by the client.**
   The end user normally enters this data in an HTML form on a Web page. However, the data could also come from an applet or a custom HTTP client program.

2. **Read the implicit HTTP request data sent by the browser.**
   Figure 2–1 shows a single arrow going from the client to the Web server (the layer where servlets and JSP execute), but there are really two varieties of data: the explicit data the end user enters in a form...
and the behind-the-scenes HTTP information. Both varieties are critical to effective development. The HTTP information includes cookies, media types and compression schemes the browser understands, and so forth.

3. **Generate the results.**
   This process may require talking to a database, executing an RMI or CORBA call, invoking a legacy application, or computing the response directly. Your real data may be in a relational database. Fine. But your database probably doesn’t speak HTTP or return results in HTML, so the Web browser can’t talk directly to the database. The same argument applies to most other applications. You need the Web middle layer to extract the incoming data from the HTTP stream, talk to the application, and embed the results inside a document.

4. **Send the explicit data (i.e., the document) to the client.**
   This document can be sent in a variety of formats, including text (HTML), binary (GIF images), or even a compressed format like gzip that is layered on top of some other underlying format.

5. **Send the implicit HTTP response data.**
   Figure 2–1 shows a single arrow going from the Web middle layer (the servlet or JSP page) to the client. But, there are really two varieties of data sent: the document itself and the behind-the-scenes HTTP information. Both varieties are critical to effective development. Sending HTTP response data involves telling the browser or other client what type of document is being returned (e.g., HTML), setting cookies and caching parameters, and other such tasks.

Many client requests can be satisfied by prebuilt documents, and the server would handle these requests without invoking servlets. In many cases, however, a static result is not sufficient, and a page needs to be generated for each request. There are a number of reasons why Web pages need to be built on-the-fly like this:

- **The Web page is based on data sent by the client.**
  For instance, the results page from search engines and order-confirmation pages at online stores are specific to particular user requests. Just remember that the user submits two kinds of data: explicit (i.e., HTML form data) and implicit (i.e., HTTP request headers). Either kind of input can be used to build the output page. In particular, it is quite common to build a user-specific page based on a cookie value.

- **The Web page is derived from data that changes frequently.**
  For example, a weather report or news headlines site might build the pages dynamically, perhaps returning a previously built page if that page is still up to date.

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
2.1 The Advantages of Servlets Over “Traditional” CGI

Java servlets are more efficient, easier to use, more powerful, more portable, safer, and cheaper than traditional CGI and many alternative CGI-like technologies.

Efficient

With traditional CGI, a new process is started for each HTTP request. If the CGI program itself is relatively short, the overhead of starting the process can dominate the execution time. With servlets, the Java virtual machine stays running and handles each request with a lightweight Java thread, not a heavyweight operating system process. Similarly, in traditional CGI, if there are \( N \) requests to the same CGI program, the code for the CGI program is loaded into memory \( N \) times. With servlets, however, there would be \( N \) threads, but only a single copy of the servlet class would be loaded. This approach reduces server memory requirements and saves time by instantiating fewer objects. Finally, when a CGI program finishes handling a request, the program terminates. This approach makes it difficult to cache computations, keep database connections open, and perform other optimizations that rely on persistent data. Servlets, however, remain in memory even after they complete a response, so it is straightforward to store arbitrarily complex data between client requests.

Convenient

Servlets have an extensive infrastructure for automatically parsing and decoding HTML form data, reading and setting HTTP headers, handling cookies, tracking sessions, and many other such high-level utilities. Besides, you already know the Java programming language. Why learn Perl too? You’re already convinced that Java

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
technology makes for more reliable and reusable code than does Visual Basic, VBScript, or C++. Why go back to those languages for server-side programming?

**Powerful**

Servlets support several capabilities that are difficult or impossible to accomplish with regular CGI. Servlets can talk directly to the Web server, whereas regular CGI programs cannot, at least not without using a server-specific API. Communicating with the Web server makes it easier to translate relative URLs into concrete path names, for instance. Multiple servlets can also share data, making it easy to implement database connection pooling and similar resource-sharing optimizations. Servlets can also maintain information from request to request, simplifying techniques like session tracking and caching of previous computations.

**Portable**

Servlets are written in the Java programming language and follow a standard API. Servlets are supported directly or by a plug-in on virtually every major Web server. Consequently, servlets written for, say, iPlanet Enterprise Server can run virtually unchanged on Apache, Microsoft Internet Information Server (IIS), IBM WebSphere, or StarNine WebStar. They are part of the Java 2 Platform, Enterprise Edition (J2EE; see http://java.sun.com/j2ee/), so industry support for servlets is becoming even more pervasive.

**Secure**

One of the main sources of vulnerabilities in traditional CGI stems from the fact that the programs are often executed by general-purpose operating system shells. So, the CGI programmer must be careful to filter out characters such as backquotes and semicolons that are treated specially by the shell. Implementing this precaution is harder than one might think, and weaknesses stemming from this problem are constantly being uncovered in widely used CGI libraries.

A second source of problems is the fact that some CGI programs are processed by languages that do not automatically check array or string bounds. For example, in C and C++ it is perfectly legal to allocate a 100-element array and then write into the 999th “element,” which is really some random part of program memory. So, programmers who forget to perform this check open up their system to deliberate or accidental buffer overflow attacks.

Servlets suffer from neither of these problems. Even if a servlet executes a system call (e.g., with `Runtime.exec` or JNI) to invoke a program on the local operating system, it does not use a shell to do so. And, of course, array bounds checking and other memory protection features are a central part of the Java programming language.

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
There are a number of free or very inexpensive Web servers that are good for development use or deployment of low- or medium-volume Web sites. Thus, with servlets and JSP you can start with a free or inexpensive server and migrate to more expensive servers with high-performance capabilities or advanced administration utilities only after your project meets initial success. This is in contrast to many of the other CGI alternatives, which require a significant initial investment for the purchase of a proprietary package.

### 2.2 Basic Servlet Structure

Listing 2.1 outlines a basic servlet that handles GET requests. GET requests, for those unfamiliar with HTTP, are the usual type of browser requests for Web pages. A browser generates this request when the user enters a URL on the address line, follows a link from a Web page, or submits an HTML form that either does not specify a METHOD or specifies METHOD="GET". Servlets can also easily handle POST requests, which are generated when someone submits an HTML form that specifies METHOD="POST". For details on using HTML forms, see Chapter 16 of Core Servlets and JavaServer Pages (available in PDF at http://www.moreservlets.com).

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

public class ServletTemplate extends HttpServlet {
    public void doGet(HttpServletRequest request,
                        HttpServletResponse response)
            throws ServletException, IOException {

        // Use "request" to read incoming HTTP headers
        // (e.g., cookies) and query data from HTML forms.

        // Use "response" to specify the HTTP response status
        // code and headers (e.g. the content type, cookies).

        PrintWriter out = response.getWriter();
        // Use "out" to send content to browser.
    }
}
```

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
To be a servlet, a class should extend HttpServlet and override doGet or doPost, depending on whether the data is being sent by GET or by POST. If you want a servlet to take the same action for both GET and POST requests, simply have doGet call doPost, or vice versa.

Both doGet and doPost take two arguments: an HttpServletRequest and an HttpServletResponse. The HttpServletRequest has methods by which you can find out about incoming information such as form (query) data, HTTP request headers, and the client's hostname. The HttpServletResponse lets you specify outgoing information such as HTTP status codes (200, 404, etc.) and response headers (Content-Type, Set-Cookie, etc.). Most importantly, it lets you obtain a PrintWriter with which you send the document content back to the client. For simple servlets, most of the effort is spent in println statements that generate the desired page. Form data, HTTP request headers, HTTP responses, and cookies are all discussed in the following sections.

Since doGet and doPost throw two exceptions, you are required to include them in the declaration. Finally, you must import classes in java.io (for PrintWriter, etc.), javax.servlet (for HttpServlet, etc.), and javax.servlet.http (for HttpServletRequest and HttpServletResponse).

### A Servlet That Generates Plain Text

Listing 2.2 shows a simple servlet that outputs plain text, with the output shown in Figure 2–2. Before we move on, it is worth spending some time reviewing the process of installing, compiling, and running this simple servlet. See Chapter 1 (Server Setup and Configuration) for a much more detailed description of the process.

First, be sure that your server is set up properly as described in Section 1.4 (Test the Server) and that your CLASSPATH refers to the necessary three entries (the JAR file containing the javax.servlet classes, your development directory, and "."), as described in Section 1.6 (Set Up Your Development Environment).

Second, type "javac HelloWorld.java" or tell your development environment to compile the servlet (e.g., by clicking Build in your IDE or selecting Compile from the emacs JDE menu). This will compile your servlet to create HelloWorld.class.

Third, move HelloWorld.class to the directory that your server uses to store servlets (usually install_dir/.../WEB-INF/classes—see Section 1.7). Alternatively, you can use one of the techniques of Section 1.8 (Establish a Simplified Deployment Method) to automatically place the class files in the appropriate location.

Finally, invoke your servlet. This last step involves using either the default URL of http://host/servlet/ServletName or a custom URL defined in the web.xml file as described in Section 5.3 (Assigning Names and Custom URLs). Figure 2–2 shows the servlet being accessed by means of the default URL, with the server running on the local machine.

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
A Servlet That Generates HTML

Most servlets generate HTML, not plain text as in the previous example. To build HTML, you need two additional steps:

1. Tell the browser that you’re sending back HTML.
2. Modify the println statements to build a legal Web page.

You accomplish the first step by setting the HTTP Content-Type response header. In general, headers are set by the setHeader method of HttpServletResponse, but setting the content type is such a common task that there is also a special setContentType method just for this purpose. The way to designate HTML is with a type of text/html, so the code would look like this:

```java
response.setContentType("text/html");
```

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
Although HTML is the most common type of document that servlets create, it is not unusual for servlets to create other document types. For example, it is quite common to use servlets to generate GIF images (content type image/gif) and Excel spreadsheets (content type application/vnd.ms-excel).

Don’t be concerned if you are not yet familiar with HTTP response headers; they are discussed in Section 2.8. Note that you need to set response headers before actually returning any of the content with the PrintWriter. That’s because an HTTP response consists of the status line, one or more headers, a blank line, and the actual document, in that order. The headers can appear in any order, and servlets buffer the headers and send them all at once, so it is legal to set the status code (part of the first line returned) even after setting headers. But servlets do not necessarily buffer the document itself, since users might want to see partial results for long pages. Servlet engines are permitted to partially buffer the output, but the size of the buffer is left unspecified. You can use the getBufferSize method of HttpServletResponse to determine the size, or you can use setBufferSize to specify it. You can set headers until the buffer fills up and is actually sent to the client. If you aren’t sure whether the buffer has been sent, you can use the isCommitted method to check. Even so, the simplest approach is to simply put the setContentType line before any of the lines that use the PrintWriter.

**Core Approach**

*Always set the content type before transmitting the actual document.*

The second step in writing a servlet that builds an HTML document is to have your println statements output HTML, not plain text. Listing 2.3 shows HelloServlet.java, the sample servlet used in Section 1.7 to verify that the server is functioning properly. As Figure 2–3 illustrates, the browser formats the result as HTML, not as plain text.

**Listing 2.3  HelloServlet.java**

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

public class HelloServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        
```

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
2.2 Basic Servlet Structure

Listing 2.3  HelloServlet.java (continued)

```java
public class HelloServlet
{
    public void doGet(HttpServletRequest req, HttpServletResponse res)
    {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        String docType = "<!DOCTYPE HTML PUBLIC ";
        out.println(docType + "-//W3C//DTD HTML 4.0 " + " Transitional//EN">\n";
        out.println("<HTML>
" + "<HEAD><TITLE>Hello</TITLE></HEAD>
" + "<BODY BGCOLOR="#FDF5E6">
" + "<H1>Hello</H1>
" + "</BODY></HTML>");
    }
}
```

Figure 2–3  Result of HelloServlet.

Servlet Packaging

In a production environment, multiple programmers can be developing servlets for the same server. So, placing all the servlets in the same directory results in a massive, hard-to-manage collection of classes and risks name conflicts when two developers accidentally choose the same servlet name. Packages are the natural solution to this problem. As we’ll see in Chapter 4, even the use of Web applications does not obviate the need for packages.

When you use packages, you need to perform the following two additional steps.

1. **Move the files to a subdirectory that matches the intended package name.** For example, I’ll use the moreservlets package for most of the rest of the servlets in this book. So, the class files need to go in a subdirectory called moreservlets.

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/.
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
2. **Insert a package statement in the class file.** For example, to place a class in a package called `somePackage`, the class should be in the `somePackage` directory and the *first* non-comment line of the file should read

```java
package somePackage;
```

For example, Listing 2.4 presents a variation of `HelloServlet` that is in the `moreservlets` package and thus the `moreservlets` directory. As discussed in Section 1.7 (Compile and Test Some Simple Servlets), the class file should be placed in `install_dir/webapps/ROOT/WEB-INF/classes/moreservlets` for Tomcat, in `install_dir/servers/default/default-app/WEB-INF/classes/moreservlets` for JRun, and in `install_dir/Servlets/moreservlets` for ServletExec.

Figure 2–4 shows the servlet accessed by means of the default URL.

**Listing 2.4 HelloServlet2.java**

```java
package moreservlets;

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

/** Simple servlet used to test the use of packages. */
public class HelloServlet2 extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String docType = "<!DOCTYPE HTML PUBLIC \
                         "/-//W3C//DTD HTML 4.0 " +
                         " Transitional//EN\">\n"
            out.println(docType +
               "<HTML>
               <HEAD><TITLE>Hello (2)</TITLE></HEAD>
               <BODY BGCOLOR="#FDF5E6">
               <H1>Hello (2)</H1>
               </BODY></HTML>");
    }
}
```

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
2.2 Basic Servlet Structure

Simple HTML-Building Utilities

As you probably already know, an HTML document is structured as follows:

```html
<!DOCTYPE ...>
<html>
<head><title>...</title>...</head>
<body ...>...</body>
</html>
```

When using servlets to build the HTML, you might be tempted to omit part of this structure, especially the `DOCTYPE` line, noting that virtually all major browsers ignore it even though the HTML 3.2 and 4.0 specifications require it. I strongly discourage this practice. The advantage of the `DOCTYPE` line is that it tells HTML validators which version of HTML you are using so they know which specification to check your document against. These validators are valuable debugging services, helping you catch HTML syntax errors that your browser guesses well on but that other browsers will have trouble displaying.

The two most popular online validators are the ones from the World Wide Web Consortium (http://validator.w3.org/) and from the Web Design Group (http://www.htmlhelp.com/tools/validator/). They let you submit a URL, then they retrieve the page, check the syntax against the formal HTML specification, and report any errors to you. Since, to a visitor, a servlet that generates HTML looks exactly like a regular Web page, it can be validated in the normal manner unless it requires `POST` data to return its result. Since `GET` data is attached to the URL, you can even send the validators a URL that includes `GET` data. If the servlet is available only inside your corporate firewall, simply run it, save the HTML to disk, and choose the validator's File Upload option.

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/.
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
Core Approach

Use an HTML validator to check the syntax of pages that your servlets generate.

Admittedly, it is a bit cumbersome to generate HTML with `println` statements, especially long tedious lines like the `DOCTYPE` declaration. Some people address this problem by writing detailed HTML-generation utilities, then use the utilities throughout their servlets. I’m skeptical of the usefulness of such an extensive library. First and foremost, the inconvenience of generating HTML programmatically is one of the main problems addressed by JavaServer Pages. Second, HTML generation routines can be cumbersome and tend not to support the full range of HTML attributes (CLASS and ID for style sheets, JavaScript event handlers, table cell background colors, and so forth).

Despite the questionable value of a full-blown HTML generation library, if you find you’re repeating the same constructs many times, you might as well create a simple utility file that simplifies those constructs. For standard servlets, two parts of the Web page (DOCTYPE and HEAD) are unlikely to change and thus could benefit from being incorporated into a simple utility file. These are shown in Listing 2.5, with Listing 2.6 showing a variation of HelloServlet that makes use of this utility. I’ll add a few more utilities throughout the chapter.

Listing 2.5  morereservlets/ServletUtilities.java

```java
package morereservlets;

import javax.servlet.*;
import javax.servlet.http.*;

/** Some simple time savers. Note that most are static methods. */

public class ServletUtilities {

  public static final String DOCTYPE =
      "<!DOCTYPE HTML PUBLIC " +
      "-//W3C//DTD HTML 4.0 " +
      " Transitional//EN">";

  public static String headWithTitle(String title) {
    return(DOCTYPE + "\n" +
      "<HTML>\n" +
      " <HEAD><TITLE>" + title + "</TITLE></HEAD>\n") ;
  }

  ...
}
```

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
2.2 Basic Servlet Structure

After you compile HelloServlet3.java (which results in ServletUtilities.java being compiled automatically), you need to move the two class files to the moreServlets subdirectory of the server’s default deployment location. If you get an “Unresolved symbol” error when compiling HelloServlet3.java, go back and review the CLASSPATH settings described in Section 1.6 (Set Up Your Development Environment). If you don’t know where to put the class files, review Sections 1.7 and 1.9. Figure 2–5 shows the result when the servlet is invoked with the default URL.

Figure 2–5  Result of HelloServlet3.
2.3 The Servlet Life Cycle

In Section 2.1 (The Advantages of Servlets Over “Traditional” CGI), I referred to the fact that only a single instance of a servlet gets created, with each user request resulting in a new thread that is handed off to doGet or doPost as appropriate. I'll now be more specific about how servlets are created and destroyed, and how and when the various methods are invoked. I give a quick summary here, then elaborate in the following subsections.

When the servlet is first created, its init method is invoked, so init is where you put one-time setup code. After this, each user request results in a thread that calls the service method of the previously created instance. Multiple concurrent requests normally result in multiple threads calling service simultaneously, although your servlet can implement a special interface (SingleThreadModel) that stipulates that only a single thread is permitted to run at any one time. The service method then calls doGet, doPost, or another doXxx method, depending on the type of HTTP request it received. Finally, when the server decides to unload a servlet, it first calls the servlet's destroy method.

The init Method

The init method is called when the servlet is first created; it is not called again for each user request. So, it is used for one-time initializations, just as with the init method of applets. The servlet is normally created when a user first invokes a URL corresponding to the servlet, but you can also specify that the servlet be loaded when the server is first started (see Section 5.5, “Initializing and Preloading Servlets and JSP Pages”).

The init method definition looks like this:

```java
public void init() throws ServletException {
    // Initialization code...
}
```

One of the most common tasks that init performs is reading server-specific initialization parameters. For example, the servlet might need to know about database settings, password files, server-specific performance parameters, hit count files, or serialized cookie data from previous requests. Initialization parameters are particularly valuable because they let the servlet deployer (e.g., the server administrator), not just the servlet author, customize the servlet.

To read initialization parameters, you first obtain a ServletConfig object by means of getServletConfig, then call getInitParameter on the result. Here is an example:

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
public void init() throws ServletException {
    ServletConfig config = getServletConfig();
    String param1 = config.getInitParameter("parameter1");
}

Notice two things about this code. First, the init method uses getServletConfig to obtain a reference to the ServletConfig object. Second, ServletConfig has a getInitParameter method with which you can look up initialization parameters associated with the servlet. Just as with the getParameter method used in the init method of applets, both the input (the parameter name) and the output (the parameter value) are strings.

You read initialization parameters by calling the getInitParameter method of ServletConfig. But how do you set them? That's the job of the web.xml file, called the deployment descriptor. This file belongs in the WEB-INF directory of the Web application you are using, and it controls many aspects of servlet and JSP behavior. Many servers provide graphical interfaces that let you specify initialization parameters and control various aspects of servlet and JSP behavior. Although those interfaces are server specific, behind the scenes they use the web.xml file, and this file is completely portable. Use of web.xml is discussed in detail in Chapter 4 (Using and Deploying Web Applications) and Chapter 5 (Controlling Web Application Behavior with web.xml), but for a quick preview, web.xml contains an XML header, a DOCTYPE declaration, and a web-app element. For the purpose of initialization parameters, the web-app element should contain a servlet element with three subelements: servlet-name, servlet-class, and init-param. The servlet-name element is the name that you want to use to access the servlet. The servlet-class element gives the fully qualified (i.e., including packages) class name of the servlet, and init-param gives names and values to initialization parameters.

For example, Listing 2.7 shows a web.xml file that gives a value to the initialization parameter called parameter1 of the OriginalServlet class that is in the somePackage package. However, the initialization parameter is available only when the servlet is accessed with the registered servlet name (or a custom URL as described in Section 5.3). So, the param1 variable in the previous code snippet would have the value "First Parameter Value" when the servlet is accessed by means of http://host/servlet/SomeName, but would have the value null when the servlet is accessed by means of http://host/servlet/somePackage.OriginalServlet.

Core Warning

Initialization parameters are not available to servlets that are accessed by means of their default URL. A registered name or custom URL must be used.

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
For more information on the `web.xml` file, including new parameters available with servlets version 2.3, see Chapter 5 (Controlling Web Application Behavior with `web.xml`). For specific details on initialization parameters and a complete working example, see Section 5.5 (Initializing and Preloading Servlets and JSP Pages).

### Listing 2.7 `web.xml` (Excerpt illustrating initialization parameters)

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
  "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
  <servlet>
    <servlet-name>SomeName</servlet-name>
    <servlet-class>somePackage.OriginalServlet</servlet-class>
    <init-param>
      <param-name>parameter1</param-name>
      <param-value>First Parameter Value</param-value>
    </init-param>
  </servlet>
  <!-- ... -->
</web-app>
```

### The service Method

Each time the server receives a request for a servlet, the server spawns a new thread (perhaps by reusing an idle `Thread` from a thread pool) and calls `service`. The `service` method checks the HTTP request type (GET, POST, PUT, DELETE, etc.) and calls `doGet`, `doPost`, `doPut`, `doDelete`, etc., as appropriate. A GET request results from a normal request for a URL or from an HTML form that has no `METHOD` specified. A POST request results from an HTML form that specifically lists POST as the `METHOD`. Other HTTP requests are generated only by custom clients. If you aren't familiar with HTML forms, see Chapter 16 of *Core Servlets and JavaServer Pages* (available in PDF at http://www.moreservlets.com).

Now, if you have a servlet that needs to handle both POST and GET requests identically, you may be tempted to override `service` directly rather than implementing both `doGet` and `doPost`. This is not a good idea. Instead, just have `doPost` call `doGet` (or vice versa), as below.
2.3 The Servlet Life Cycle

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

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

Although this approach takes a couple of extra lines of code, it has several advantages over directly overriding `service`. First, you can later add support for other HTTP request methods by adding `doPut`, `doTrace`, etc., perhaps in a subclass. Overriding `service` directly precludes this possibility. Second, you can add support for modification dates by adding a `getLastModified` method. Since `getLastModified` is invoked by the default `service` method, overriding `service` eliminates this option. Finally, you get automatic support for `HEAD`, `OPTION`, and `TRACE` requests.

**Core Approach**

*If your servlet needs to handle both GET and POST identically, have your doPost method call doGet, or vice versa. Don’t override service.*

**The doGet, doPost, and doXxx Methods**

These methods contain the real meat of your servlet. Ninety-nine percent of the time, you only care about GET or POST requests, so you override `doGet` and/or `doPost`. However, if you want to, you can also override `doDelete` for DELETE requests, `doPut` for PUT, `doOptions` for OPTIONS, and `doTrace` for TRACE. Recall, however, that you have automatic support for OPTIONS and TRACE.

In versions 2.1 and 2.2 of the servlet API, there is no `doHead` method. That’s because the system automatically uses the status line and header settings of `doGet` to answer HEAD requests. In version 2.3, however, `doHead` was added so that you can generate responses to HEAD requests (i.e., requests from custom clients that want just the HTTP headers, not the actual document) more quickly—without building the actual document output.

Source code for all examples in book: http://www.moreservlets.com/
JEE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
The SingleThreadModel Interface

Normally, the system makes a single instance of your servlet and then creates a new thread for each user request, with multiple concurrent threads running if a new request comes in while a previous request is still executing. This means that your doGet and doPost methods must be careful to synchronize access to fields and other shared data, since multiple threads may access the data simultaneously. If you want to prevent this multithreaded access, you can have your servlet implement the SingleThreadModel interface, as below.

```java
public class YourServlet extends HttpServlet
    implements SingleThreadModel {
    ...
}
```

If you implement this interface, the system guarantees that there is never more than one request thread accessing a single instance of your servlet. In most cases, it does so by queuing all the requests and passing them one at a time to a single servlet instance. However, the server is permitted to create a pool of multiple instances, each of which handles one request at a time. Either way, this means that you don’t have to worry about simultaneous access to regular fields (instance variables) of the servlet. You do, however, still have to synchronize access to class variables (static fields) or shared data stored outside the servlet.

Synchronous access to your servlets can significantly hurt performance (latency) if your servlet is accessed frequently. When a servlet waits for I/O, the server remains idle instead of handling pending requests. So, think twice before using the SingleThreadModel approach.

**Core Warning**

Avoid implementing SingleThreadModel for high-traffic servlets. Use explicit synchronized blocks instead.

The destroy Method

The server may decide to remove a previously loaded servlet instance, perhaps because it is explicitly asked to do so by the server administrator, or perhaps because the servlet is idle for a long time. Before it does, however, it calls the servlet’s destroy method. This method gives your servlet a chance to close database connections, halt background threads, write cookie lists or hit counts to disk, and perform other such cleanup activities. Be aware, however, that it is possible for the Web
server to crash. So, don’t count on destroy as the only mechanism for saving state to disk. Activities like hit counting or accumulating lists of cookie values that indicate special access should also proactively write their state to disk periodically.

2.4 The Client Request: Form Data

One of the main motivations for building Web pages dynamically is to base the result upon query data submitted by the user. This section briefly shows you how to access that data. More details are provided in Chapter 3 of Core Servlets and JavaServer Pages (available in PDF at http://www.moreservlets.com).

If you’ve ever used a search engine, visited an online bookstore, tracked stocks on the Web, or asked a Web-based site for quotes on plane tickets, you’ve probably seen funny-looking URLs like http://host/path?user=Marty+Hall&origin=bwi&dest=nrt. The part after the question mark (i.e., user=Marty+Hall&origin=bwi&dest=nrt) is known as form data (or query data) and is the most common way to get information from a Web page to a server-side program. Form data can be attached to the end of the URL after a question mark (as above) for GET requests or sent to the server on a separate line for POST requests. If you’re not familiar with HTML forms, see Chapter 16 of Core Servlets and JavaServer Pages (in PDF at http://www.moreservlets.com) for details on how to build forms that collect and transmit data of this sort.

Reading Form Data from CGI Programs

Extracting the needed information from form data is traditionally one of the most tedious parts of CGI programming. First, you have to read the data one way for GET requests (in traditional CGI, this is usually through the QUERY_STRING environment variable) and a different way for POST requests (by reading the standard input in traditional CGI). Second, you have to chop the pairs at the ampersands, then separate the parameter names (left of the equal signs) from the parameter values (right of the equal signs). Third, you have to URL-decode the values. Alphanumeric characters are sent unchanged, but spaces are converted to plus signs and other characters are converted to %XX where XX is the ASCII (or ISO Latin-1) value of the character, in hex.

Reading Form Data from Servlets

One of the nice features of servlets is that all the form parsing is handled automatically. You simply call the getParameter method of HttpServletRequest, supplying the case-sensitive parameter name as an argument. You use getParameter exactly the same way when the data is sent by GET as you do when it is sent by POST.

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
Chapter 2: A Fast Introduction to Basic Servlet Programming

The servlet knows which request method was used and automatically does the right thing behind the scenes. The return value is a String corresponding to the URL-decoded value of the first occurrence of that parameter name. An empty String is returned if the parameter exists but has no value, and null is returned if there is no such parameter in the request.

Technically, it is legal for a single HTML form to use the same parameter name twice, and in fact this situation really occurs when you use SELECT elements that allow multiple selections (see Section 16.6 of Core Servlets and JavaServer Pages). If the parameter could potentially have more than one value, you should call getParameterValues (which returns an array of strings) instead of getParameter (which returns a single string). The return value of getParameterValues is null for nonexistent parameter names and is a one-element array when the parameter has only a single value.

Parameter names are case sensitive, so, for example, request.getParameter("Param1") and request.getParameter("param1") are not interchangeable.

Core Note

The values supplied to getParameter and getParameterValues are case sensitive.

Finally, although most real servlets look for a specific set of parameter names, for debugging purposes it is sometimes useful to get a full list. Use getParameterNames to get this list in the form of an Enumeration, each entry of which can be cast to a String and used in a getParameter or getParameterValues call. Just note that the HttpServletRequest API does not specify the order in which the names appear within that Enumeration.

Example: Reading Three Explicit Parameters

Listing 2.8 presents a simple servlet called ThreeParams that reads form data parameters named param1, param2, and param3 and places their values in a bulleted list. Listing 2.9 shows an HTML form that collects user input and sends it to this servlet. By use of an ACTION URL that begins with a slash (e.g., /servlet/moreservlets.ThreeParams), the form can be installed anywhere in the server's Web document hierarchy; there need not be any particular association between the directory containing the form and the servlet installation directory. When you use Web applications, HTML files (and images and JSP pages) go in the directory above the one containing the WEB-INF directory; see Section 4.2 (Structure of a Web Application) for details. The directory for HTML files that are not part of an explicit Web

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
application varies from server to server. As described in Section 1.5 (Try Some Simple HTML and JSP Pages) HTML and JSP pages go in install_dir/webapps/ROOT for Tomcat, in install_dir/servers/default/default-app for JRun, and in install_dir/public_html for ServletExec. For other servers, see the appropriate server documentation.

Also note that the ThreeParams servlet reads the query data after it starts generating the page. Although you are required to specify response settings before beginning to generate the content, there is no requirement that you read the request parameters at any particular time.

### Listing 2.8 ThreeParams.java

```java
package moreservlets;

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

/** Simple servlet that reads three parameters from the form data.
 */
public class ThreeParams extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Reading Three Request Parameters"
        out.println(ServletUtilities.headWithTitle(title) +
                    "<BODY BGCOLOR="#FDF5E6">
                    <H1 ALIGN="CENTER">" + title + "</H1>" +
                    "\n" +
                    "<UL>
                        <LI><B>param1</B>: " +
                        request.getParameter("param1") + "\n" +
                        "<LI><B>param2</B>: " +
                        request.getParameter("param2") + "\n" +
                        "<LI><B>param3</B>: " +
                        request.getParameter("param3") + "\n" +
                    "</UL>" +
                    "</BODY>" +
                    "</HTML>" +
    }
}
```

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
Figure 2–6 shows the HTML form after the user enters the home directories of three famous Internet personalities (OK, two famous Internet personalities). Figure 2–7 shows the result of the form submission. Note that, although the form contained ~, a non-alphanumeric character that was transmitted by use of its hex-encoded Latin-1 value (%7E), the servlet had to do nothing special to get the value as it was typed into the HTML form. This conversion (called URL decoding) is done automatically. Servlet authors simply specify the parameter name as it appears in the HTML source code and get back the parameter value as it was entered by the end user: a big improvement over CGI and many alternatives to servlets and JSP.

If you’re accustomed to the traditional CGI approach where you read POST data through the standard input, you should note that it is possible to do the same thing with servlets by calling getReader or getInputStream on the HttpServletRequest and then using that stream to obtain the raw input. This is a bad idea for regular parameters; getParameter is simpler and yields results that are parsed and URL-decoded. However, reading the raw input might be of use for uploaded files or POST data being sent by custom clients. Note, however, that if you read the POST data in this manner, it might no longer be found by getParameter.

Core Warning

Do not use getParameter when you also call getInputStream and read the raw servlet input.

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
Filtering Query Data

In the previous example, we read the `param1`, `param2`, and `param3` request parameters and inserted them verbatim into the page being generated. This is not necessarily safe, since the request parameters might contain HTML characters such as “<” that could disrupt the rest of the page processing, causing some of the subsequent tags to be interpreted incorrectly. For an example, see Section 3.6 of Core Servlets and JavaServer Pages (available in PDF at http://www.moreservlets.com). A safer approach is to filter out the HTML-specific characters before inserting the values into the page. Listing 2.10 shows a static filter method that accomplishes this task.

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
2.5 The Client Request: HTTP Request Headers

One of the keys to creating effective servlets is understanding how to manipulate the HyperText Transfer Protocol (HTTP). Getting a thorough grasp of this protocol is not an esoteric, theoretical concept, but rather a practical issue that can have an

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
immediate impact on the performance and usability of your servlets. This section discusses the HTTP information that is sent from the browser to the server in the form of request headers. It explains a few of the most important HTTP 1.1 request headers, summarizing how and why they would be used in a servlet. For more details and examples, see Chapter 4 of Core Servlets and JavaServer Pages (available in PDF at http://www.moreservlets.com).

Note that HTTP request headers are distinct from the form (query) data discussed in the previous section. Form data results directly from user input and is sent as part of the URL for GET requests and on a separate line for POST requests. Request headers, on the other hand, are indirectly set by the browser and are sent immediately following the initial GET or POST request line. For instance, the following example shows an HTTP request that might result from a user submitting a book-search request to a servlet at http://www.somebookstore.com/servlet/Search. The request includes the headers Accept, Accept-Encoding, Connection, Cookie, Host, Referer, and User-Agent, all of which might be important to the operation of the servlet, but none of which can be derived from the form data or deduced automatically: the servlet needs to explicitly read the request headers to make use of this information.

GET /servlet/Search?keywords=servlets+jsp HTTP/1.1
Accept: image/gif, image/jpg, */*
Accept-Encoding: gzip
Connection: Keep-Alive
Cookie: userID=id456578
Host: www.somebookstore.com
Referer: http://www.somebookstore.com/findbooks.html
User-Agent: Mozilla/4.7 [en] (Win98; U)

Reading Request Headers from Servlets

Reading headers is straightforward; just call the getHeader method of HttpServletRequest, which returns a String if the specified header was supplied on this request, null otherwise. Header names are not case sensitive. So, for example, request.getHeader("Connection") is interchangeable with request.getHeader("connection").

Although getHeader is the general-purpose way to read incoming headers, a few headers are so commonly used that they have special access methods in HttpServletRequest. Following is a summary.

- **getCookies**
  The getCookies method returns the contents of the Cookie header, parsed and stored in an array of Cookie objects. This method is discussed in more detail in Section 2.9 (Cookies).
Chapter 2: A Fast Introduction to Basic Servlet Programming

- **getAuthType** and **getRemoteUser**
  The `getAuthType` and `getRemoteUser` methods break the Authorization header into its component pieces.

- **getContentType**
  The `getContentType` method returns the value of the Content-Type header (as a String).

- **getDateHeader** and **getIntHeader**
  The `getDateHeader` and `getIntHeader` methods read the specified headers and then convert them to Date and int values, respectively.

- **getHeaderNames**
  Rather than looking up one particular header, you can use the `getHeaderNames` method to get an `Enumeration` of all header names received on this particular request. This capability is illustrated in Listing 2.11.

- **getHeaders**
  In most cases, each header name appears only once in the request. Occasionally, however, a header can appear multiple times, with each occurrence listing a separate value. Accept-Language is one such example. You can use `getHeaders` to obtain an `Enumeration` of the values of all occurrences of the header.

Finally, in addition to looking up the request headers, you can get information on the main request line itself, also by means of methods in `HttpServletRequest`. Here is a summary of the three main methods.

- **getMethod**
  The `getMethod` method returns the main request method (normally GET or POST, but methods like HEAD, PUT, and DELETE are possible).

- **getRequestURI**
  The `getRequestURI` method returns the part of the URL that comes after the host and port but before the form data. For example, for a URL of `http://randomhost.com/servlet/search.BookSearch`, `getRequestURI` would return `"/servlet/search.BookSearch"`.

- **getProtocol**
  The `getProtocol` method returns the third part of the request line, which is generally HTTP/1.0 or HTTP/1.1. Servlets should usually check `getProtocol` before specifying response headers (Section 2.8) that are specific to HTTP 1.1.

Source code for all examples in book: http://www.moreservlets.com/

J2EE training from the author: http://courses.coreservlets.com/

Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
Example: Making a Table of All Request Headers

Listing 2.11 shows a servlet that simply creates a table of all the headers it receives, along with their associated values. It also prints out the three components of the main request line (method, URI, and protocol). Figures 2–8 and 2–9 show typical results with Netscape and Internet Explorer.

Listing 2.11  ShowRequestHeaders.java

```java
package moreservlets;

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

/** Shows all the request headers sent on this request. */
public class ShowRequestHeaders extends HttpServlet {
    public void doGet(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Servlet Example: Showing Request Headers";
        out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR="#FDF5E6">
                <H1 ALIGN="CENTER">" + title + "</H1>\n" +
                "<B>Request Method: </B>" +
                request.getMethod() + "<BR>\n" +
                "<B>Request URI: </B>" +
                request.getRequestURI() + "<BR>\n" +
                "<B>Request Protocol: </B>" +
                request.getProtocol() + "<BR>\n" +
                "<TABLE BORDER=1 ALIGN="CENTER">\n" +
                "<TR BGCOLOR="#FDF5E6">\n" +
                "<TD><H1 ALIGN="CENTER">" + title + "</H1>\n" +
                "<TH>Header Name<TH>Header Value";
        Enumeration headerNames = request.getHeaderNames();
        while(headerNames.hasMoreElements()) {
            String headerName = (String)headerNames.nextElement();
            out.println("<TR><TD>" + headerName);
            out.println("<TD>" + request.getHeader(headerName));
        }
        out.println("</TABLE>\n" + request.getHeader(headerName));
    }
}
```

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
Chapter 2: A Fast Introduction to Basic Servlet Programming

Listing 2.11 ShowRequestHeaders.java (continued)

```java
/** Let the same servlet handle both GET and POST. */

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

Servlet Example: Showing Request Headers

**Request Method**: GET  
**Request URL**: /servlet/showrequestheaders

<table>
<thead>
<tr>
<th>Header Name</th>
<th>Header Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>connection</td>
<td>Keep-Alive</td>
</tr>
<tr>
<td>user-agent</td>
<td>Mozilla 4.7 [en] (Win98, U)</td>
</tr>
<tr>
<td>host</td>
<td>localhost</td>
</tr>
<tr>
<td>accept</td>
<td>image/gif, image/jpeg, image/png, image/ico, image/x-icon, <em>/</em></td>
</tr>
<tr>
<td>accept-encoding</td>
<td>gzip</td>
</tr>
<tr>
<td>accept-language</td>
<td>en</td>
</tr>
<tr>
<td>accept-charset</td>
<td>iso-8859-1, utf-8</td>
</tr>
</tbody>
</table>

Figure 2–8  Request headers sent by Netscape 4.7 on Windows 98.

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
2.5 The Client Request: HTTP Request Headers

The Client Request: HTTP Request Headers

Understand HTTP 1.1 Request Headers

Access to the request headers permits servlets to perform a number of optimizations and to provide a number of features not otherwise possible. This subsection summarizes the headers most often used by servlets; more details are given in Core Servlets and JavaServer Pages, Chapter 4 (in PDF at http://www.moreservlets.com). Note that HTTP 1.1 supports a superset of the headers permitted in HTTP 1.0. For additional details on these and other headers, see the HTTP 1.1 specification, given in RFC 2616. The official RFCs are archived in a number of places; your best bet is to start at http://www.rfc-editor.org/ to get a current list of the archive sites.

Accept

This header specifies the MIME types that the browser or other clients can handle. A servlet that can return a resource in more than one format can examine the Accept header to decide which format to use. For example, images in PNG format have some compression advantages over those in GIF, but only a few browsers support PNG. If you had images in both formats, a servlet could...
call request.getHeader("Accept"), check for image/png, and if it finds a match, use xxx.png filenames in all the IMG elements it generates. Otherwise, it would just use xxx.gif.

See Table 2.1 in Section 2.8 (The Server Response: HTTP Response Headers) for the names and meanings of the common MIME types.

Note that Internet Explorer 5 has a bug whereby the Accept header is not sent properly when you reload a page. It is sent properly on the original request, however.

Accept-Charset
This header indicates the character sets (e.g., ISO-8859-1) the browser can use.

Accept-Encoding
This header designates the types of encodings that the client knows how to handle. If the server receives this header, it is free to encode the page by using one of the formats specified (usually to reduce transmission time), sending the Content-Encoding response header to indicate that it has done so. This encoding type is completely distinct from the MIME type of the actual document (as specified in the Content-Type response header), since this encoding is reversed before the browser decides what to do with the content. On the other hand, using an encoding the browser doesn’t understand results in totally incomprehensible pages. Consequently, it is critical that you explicitly check the Accept-Encoding header before using any type of content encoding. Values of gzip or compress are the two most common possibilities.

Compressing pages before returning them is a valuable service because the decoding time is likely to be small compared to the savings in transmission time. See Section 9.5 where gzip compression is used to reduce download times by a factor of 10.

Accept-Language
This header specifies the client's preferred languages in case the servlet can produce results in more than one language. The value of the header should be one of the standard language codes such as en, en-us, da, etc. See RFC 1766 for details (start at http://www.rfc-editor.org/ to get a current list of the RFC archive sites).

Authorization
This header is used by clients to identify themselves when accessing password-protected Web pages. For details, see Chapters 7 and 8.
2.5 The Client Request: HTTP Request Headers

Connection
This header indicates whether the client can handle persistent HTTP connections. Persistent connections permit the client or other browser to retrieve multiple files (e.g., an HTML file and several associated images) with a single socket connection, saving the overhead of negotiating several independent connections. With an HTTP 1.1 request, persistent connections are the default, and the client must specify a value of close for this header to use old-style connections. In HTTP 1.0, a value of Keep-Alive means that persistent connections should be used.

Each HTTP request results in a new invocation of a servlet (i.e., a thread calling the servlet’s service and doXXX methods), regardless of whether the request is a separate connection. That is, the server invokes the servlet only after the server has already read the HTTP request. This means that servlets need help from the server to handle persistent connections. Consequently, the servlet’s job is just to make it possible for the server to use persistent connections, which the servlet does by setting the Content-Length response header.

Content-Length
This header is applicable only to POST requests and gives the size of the POST data in bytes. Rather than calling request.getIntHeader("Content-Length"), you can simply use request.getContentLength(). However, since servlets take care of reading the form data for you (see Section 2.4), you rarely use this header explicitly.

Cookie
This header is used to return cookies to servers that previously sent them to the browser. Never read this header directly; use request.getCookies instead. For details, see Section 2.9 (Cookies). Technically, Cookie is not part of HTTP 1.1. It was originally a Netscape extension but is now widely supported, including in both Netscape Navigator/Communicator and Microsoft Internet Explorer.

Host
In HTTP 1.1, browsers and other clients are required to specify this header, which indicates the host and port as given in the original URL. Due to request forwarding and machines that have multiple hostnames, it is quite possible that the server could not otherwise determine this information. This header is not new in HTTP 1.1, but in HTTP 1.0 it was optional, not required.

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/.
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
If-Modified-Since
This header indicates that the client wants the page only if it has been changed after the specified date. The server sends a 304 (Not Modified) header if no newer result is available. This option is useful because it lets browsers cache documents and reload them over the network only when they’ve changed. However, servlets don’t need to deal directly with this header. Instead, they should just implement the `getLastModified` method to have the system handle modification dates automatically. See Section 2.8 of Core Servlets and JavaServer Pages (available in PDF at http://www.moreservlets.com) for an example of the use of `getLastModified`.

If-Unmodified-Since
This header is the reverse of If-Modified-Since; it specifies that the operation should succeed only if the document is older than the specified date. Typically, If-Modified-Since is used for GET requests (“give me the document only if it is newer than my cached version”), whereas If-Unmodified-Since is used for PUT requests (“update this document only if nobody else has changed it since I generated it”). This header is new in HTTP 1.1.

Referer
This header indicates the URL of the referring Web page. For example, if you are at Web page 1 and click on a link to Web page 2, the URL of Web page 1 is included in the Referer header when the browser requests Web page 2. All major browsers set this header, so it is a useful way of tracking where requests come from. This capability is helpful for tracking advertisers who refer people to your site, for slightly changing content depending on the referring site, or simply for keeping track of where your traffic comes from. In the last case, most people simply rely on Web server log files, since the Referer is typically recorded there. Although the Referer header is useful, don’t rely too heavily on it since it can easily be spoofed by a custom client. Finally, note that, due to a spelling mistake by one of the original HTTP authors, this header is Referer, not the expected Referrer.

User-Agent
This header identifies the browser or other client making the request and can be used to return different content to different types of browsers. Be wary of this usage when dealing only with Web browsers; relying on a hard-coded list of browser versions and associated features can make for unreliable and hard-to-modify servlet code. Whenever possible, use something specific in the HTTP headers instead. For example, instead of trying to remember which browsers support gzip on which platforms, simply check the `Accept-Encoding` header.

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
However, the `User-Agent` header is quite useful for distinguishing among different categories of client. For example, Japanese developers might see if the `User-Agent` is an Imode cell phone (in which case you would redirect to a chtml page), a Skynet cell phone (in which case you would redirect to a wml page), or a Web browser (in which case you would generate regular HTML).

Most Internet Explorer versions list a “Mozilla” (Netscape) version first in their `User-Agent` line, with the real browser version listed parenthetically. This is done for compatibility with JavaScript, where the `User-Agent` header is sometimes used to determine which JavaScript features are supported. Also note that this header can be easily spoofed, a fact that calls into question the reliability of sites that use this header to “show” market penetration of various browser versions.

### 2.6 The Servlet Equivalent of the Standard CGI Variables

If you come to servlets with a background in traditional Common Gateway Interface (CGI) programming, you are probably used to the idea of “CGI variables.” These are a somewhat eclectic collection of information about the current request. Some are based on the HTTP request line and headers (e.g., form data), others are derived from the socket itself (e.g., the name and IP address of the requesting host), and still others are taken from server installation parameters (e.g., the mapping of URLs to actual paths).

Although it probably makes more sense to think of different sources of data (request data, server information, etc.) as distinct, experienced CGI programmers may find it useful to see the servlet equivalent of each of the CGI variables. If you don’t have a background in traditional CGI, first, count your blessings; servlets are easier to use, more flexible, and more efficient than standard CGI. Second, just skim this section, noting the parts not directly related to the incoming HTTP request. In particular, observe that you can use `getServletContext().getRealPath` to map a URI (here, URI refers to the part of the URL that comes after the host and port) to an actual path and that you can use `request.getRemoteHost()` and `request.getRemoteAddress()` to get the name and IP address of the client.

**AUTH_TYPE**

If an `Authorization` header was supplied, this variable gives the scheme specified (basic or digest). Access it with `request.getAuthType()`.
CONTENT_LENGTH
For POST requests only, this variable stores the number of bytes of data sent, as given by the Content-Length request header. Technically, since the CONTENT_LENGTH CGI variable is a string, the servlet equivalent is String.valueOf(request.getContentType()) or request.getHeader("Content-Length"). You'll probably want to just call request.getContentType(), which returns an int.

CONTENT_TYPE
CONTENT_TYPE designates the MIME type of attached data, if specified.

See Table 2.1 in Section 2.8 (The Server Response: HTTP Response Headers) for the names and meanings of the common MIME types. Access CONTENT_TYPE with request.getContentType().

DOCUMENT_ROOT
The DOCUMENT_ROOT variable specifies the real directory corresponding to the URL http://host/. Access it with getServletContext().getRealPath("/"). In older servlet specifications, you accessed this variable with request.getRealPath("/"); however, the older access method is no longer supported. Also, you can use getServletContext().getRealPath to map an arbitrary URI (i.e., URL suffix that comes after the hostname and port) to an actual path on the local machine.

HTTP_XXX_YYY
Variables of the form HTTP_HEADER_NAME were how CGI programs obtained access to arbitrary HTTP request headers. The Cookie header became HTTP_COOKIE, User-Agent became HTTP_USER_AGENT, Referer became HTTP_REFERER, and so forth. Servlets should just use request.getHeader or one of the shortcut methods described in Section 2.5 (The Client Request: HTTP Request Headers).

PATH_INFO
This variable supplies any path information attached to the URL after the address of the servlet but before the query data. For example, with http://host/servlet/moreservlets.SomeServlet/foo/bar?baz=quux, the path information is /foo/bar. Since servlets, unlike standard CGI programs, can talk directly to the server, they don't need to treat path information specially. Path information could be sent as part of the regular form data and then translated by getServletContext().getRealPath. Access the value of PATH_INFO by using request.getPathInfo().
**2.6 The Servlet Equivalent of the Standard CGI Variables**

**PATH_TRANSLATED**

PATH_TRANSLATED gives the path information mapped to a real path on the server. Again, with servlets there is no need to have a special case for path information, since a servlet can call `getServletContext().getRealPath()` to translate partial URLs into real paths. This translation is not possible with standard CGI because the CGI program runs entirely separately from the server. Access this variable by means of `request.getPathTranslated()`.

**QUERY_STRING**

For GET requests, this variable gives the attached data as a single string with values still URL-encoded. You rarely want the raw data in servlets; instead, use `request.getParameter` to access individual parameters, as described in Section 2.5 (The Client Request: HTTP Request Headers). However, if you do want the raw data, you can get it with `request.getQueryString()`.

**REMOTE_ADDR**

This variable designates the IP address of the client that made the request, as a String (e.g., "198.137.241.30"). Access it by calling `request.getRemoteAddr()`.

**REMOTE_HOST**

REMOTE_HOST indicates the fully qualified domain name (e.g., whitehouse.gov) of the client that made the request. The IP address is returned if the domain name cannot be determined. You can access this variable with `request.getRemoteHost()`.

**REMOTE_USER**

If an Authorization header was supplied and decoded by the server itself, the REMOTE_USER variable gives the user part, which is useful for session tracking in protected sites. Access it with `request.getRemoteUser()`.

**REQUEST_METHOD**

This variable stipulates the HTTP request type, which is usually GET or POST but is occasionally HEAD, PUT, DELETE, OPTIONS, or TRACE. Servlets rarely need to look up REQUEST_METHOD explicitly, since each of the request types is typically handled by a different servlet method (doGet, doPost, etc.). An exception is HEAD, which is handled automatically by the service method returning whatever headers and status codes the doGet method would use. Access this variable by means of `request.getMethod()`.

---

Source code for all examples in book: http://www.moreservlets.com/
JEE training from the author: http://courses.coreservlets.com/.
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
SCRIPT_NAME
This variable specifies the path to the server-side program (i.e., the servlet in our case), relative to the server's root directory. It can be accessed through request.getServletPath().

SERVER_NAME
SERVER_NAME gives the host name of the server machine. It can be accessed by means of request.getServerName().

SERVER_PORT
This variable stores the port the server is listening on. Technically, the servlet equivalent is String.valueOf(request.getServerPort()), which returns a String. You'll usually just want request.getServerPort(), which returns an int.

SERVER_PROTOCOL
The SERVER_PROTOCOL variable indicates the protocol name and version used in the request line (e.g., HTTP/1.0 or HTTP/1.1). Access it by calling request.getProtocol().

SERVER_SOFTWARE
This variable gives identifying information about the Web server. Access it with getServletContext().getServerInfo().

2.7 The Server Response: HTTP Status Codes

When a Web server responds to a request from a browser or other Web client, the response typically consists of a status line, some response headers, a blank line, and the document. Here is a minimal example:

HTTP/1.1 200 OK
Content-Type: text/plain

Hello World

The status line consists of the HTTP version (HTTP/1.1 in the example above), a status code (an integer; 200 in the example), and a very short message corresponding to the status code (OK in the example). In most cases, all of the headers are optional except for Content-Type, which specifies the MIME type of the document that
follows. Although most responses contain a document, some don’t. For example, responses to HEAD requests should never include a document, and a variety of status codes essentially indicate failure and either don’t include a document or include only a short error-message document.

Servlets can perform a variety of important tasks by manipulating the status line and the response headers. For example, they can forward the user to other sites; indicate that the attached document is an image, Adobe Acrobat file, or HTML file; tell the user that a password is required to access the document; and so forth. This section briefly summarizes the most important status codes and what can be accomplished with them; see Chapter 6 of Core Servlets and JavaServer Pages (in PDF at http://www.moreservlets.com) for more details. The following section discusses the response headers.

## Specifying Status Codes

As just described, the HTTP response status line consists of an HTTP version, a status code, and an associated message. Since the message is directly associated with the status code and the HTTP version is determined by the server, all a servlet needs to do is to set the status code. A code of 200 is set automatically, so servlets don’t usually need to specify a status code at all. When they do set a code, they do so with the `setStatus` method of `HttpServletResponse`. If your response includes a special status code and a document, be sure to call `setStatus` before actually returning any of the content with the `PrintWriter`. That’s because an HTTP response consists of the status line, one or more headers, a blank line, and the actual document, in that order. As discussed in Section 2.2 (Basic Servlet Structure), servlets do not necessarily buffer the document (version 2.1 servlets never do so), so you have to either set the status code before first using the `PrintWriter` or carefully check that the buffer hasn’t been flushed and content actually sent to the browser.

### Core Approach

Set status codes **before** sending any document content to the client.

The `setStatus` method takes an int (the status code) as an argument, but instead of using explicit numbers, for clarity and reliability use the constants defined in `HttpServletResponse`. The name of each constant is derived from the standard HTTP 1.1 message for each constant, all upper case with a prefix of `SC` (for `Status Code`) and spaces changed to underscores. Thus, since the message for 404 is Not Found, the equivalent constant in `HttpServletResponse` is `SC_NOT_FOUND`. There are two exceptions, however. The constant for code 302 is derived from the HTTP 1.0 message (Moved Temporarily), not the HTTP 1.1 message (Found), and the constant for code 307 (Temporary Redirect) is missing altogether.

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/.
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
Although the general method of setting status codes is simply to call
response.setStatus(int), there are two common cases where a shortcut
method in HttpServletResponse is provided. Just be aware that both of these
methods throw IOException, whereas setStatus does not.

- **public void sendError(int code, String message)**
The sendError method sends a status code (usually 404) along with
a short message that is automatically formatted inside an HTML
document and sent to the client.

- **public void sendRedirect(String url)**
The sendRedirect method generates a 302 response along with a
Location header giving the URL of the new document. With
servlets version 2.1, this must be an absolute URL. In version 2.2 and
2.3, either an absolute or a relative URL is permitted; the system
automatically translates relative URLs into absolute ones before
putting them in the Location header.

Setting a status code does not necessarily mean that you don’t need to return a
document. For example, although most servers automatically generate a small File
Not Found message for 404 responses, a servlet might want to customize this
response. Again, remember that if you do send output, you have to call
setStatus or sendError first.

**HTTP 1.1 Status Codes**

In this subsection I describe the most important status codes available for use in
servlets talking to HTTP 1.1 clients, along with the standard message associated with
each code. A good understanding of these codes can dramatically increase the capa-
bilities of your servlets, so you should at least skim the descriptions to see what
options are at your disposal. You can come back for details when you are ready to
make use of some of the capabilities.

The complete HTTP 1.1 specification is given in RFC 2616. In general, you can
access RFCs online by going to http://www.rfc-editor.org/ and following the links to
the latest RFC archive sites, but since this one came from the World Wide Web Con-
sortium, you can just go to http://www.w3.org/Protocols/. Codes that are new in
HTTP 1.1 are noted, since some browsers support only HTTP 1.0. You should only
send the new codes to clients that support HTTP 1.1, as verified by checking
request.getRequestProtocol.

The rest of this section describes the specific status codes available in HTTP 1.1.
These codes fall into five general categories:

- **100–199**
  Codes in the 100s are informational, indicating that the client should
  respond with some other action.
2.7 The Server Response: HTTP Status Codes

- **200–299**
  Values in the 200s signify that the request was successful.

- **300–399**
  Values in the 300s are used for files that have moved and usually include a Location header indicating the new address.

- **400–499**
  Values in the 400s indicate an error by the client.

- **500–599**
  Codes in the 500s signify an error by the server.

The constants in HttpServletResponse that represent the various codes are derived from the standard messages associated with the codes. In servlets, you usually refer to status codes only by means of these constants. For example, you would use response.setStatus(response.SC_NO_CONTENT) rather than response.setStatus(204), since the latter is unclear to readers and is prone to typographical errors. However, you should note that servers are allowed to vary the messages slightly, and clients pay attention only to the numeric value. So, for example, you might see a server return a status line of HTTP/1.1 200 Document Follows instead of HTTP/1.1 200 OK.

100 (Continue)
If the server receives an Expect request header with a value of 100-continue, it means that the client is asking if it can send an attached document in a follow-up request. In such a case, the server should either respond with status 100 (SC_CONTINUE) to tell the client to go ahead or use 417 (Expectation Failed) to tell the browser it won’t accept the document. This status code is new in HTTP 1.1.

200 (OK)
A value of 200 (SC_OK) means that everything is fine. The document follows for GET and POST requests. This status is the default for servlets; if you don’t use setStatus, you’ll get 200.

202 (Accepted)
A value of 202 (SC_ACCEPTED) tells the client that the request is being acted upon, but processing is not yet complete.

204 (No Content)
A status code of 204 (SC_NO_CONTENT) stipulates that the browser should continue to display the previous document because no new document is available. This behavior is useful if the user periodically reloads a page by pressing the Reload button, and you can determine that the previous page is already up-to-date.

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/.
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
205 (Reset Content)

A value of 205 (SC_RESET_CONTENT) means that there is no new document, but the browser should reset the document view. This status code instructs browsers to clear form fields. It is new in HTTP 1.1.

301 (Moved Permanently)

The 301 (SC_MOVED_PERMANENTLY) status indicates that the requested document is elsewhere; the new URL for the document is given in the Location response header. Browsers should automatically follow the link to the new URL.

302 (Found)

This value is similar to 301, except that in principle the URL given by the Location header should be interpreted as a temporary replacement, not a permanent one. In practice, most browsers treat 301 and 302 identically. Note: In HTTP 1.0, the message was Moved Temporarily instead of Found, and the constant in HttpServletResponse is SC_MOVED_TEMPORARILY, not the expected SC_FOUND.

Core Note

The constant representing 302 is SC_MOVED_TEMPORARILY, not SC_FOUND.

Status code 302 is useful because browsers automatically follow the reference to the new URL given in the Location response header. It is so useful, in fact, that there is a special method for it, sendRedirect. Using response.sendRedirect(url) has a couple of advantages over using response.setStatus(response.SC_MOVED_TEMPORARILY) and response.setHeader("Location", url). First, it is shorter and easier. Second, with sendRedirect, the servlet automatically builds a page containing the link to show to older browsers that don’t automatically follow redirects. Finally, with version 2.2 and 2.3 of servlets, sendRedirect can handle relative URLs, automatically translating them into absolute ones.

Technically, browsers are only supposed to automatically follow the redirection if the original request was GET. For details, see the discussion of the 307 status code.

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
303 (See Other)
The 303 (SC_SEE_OTHER) status is similar to 301 and 302, except that if the original request was POST, the new document (given in the Location header) should be retrieved with GET. This code is new in HTTP 1.1.

304 (Not Modified)
When a client has a cached document, it can perform a conditional request by supplying an If-Modified-Since header to indicate that it wants the document only if it has been changed since the specified date. A value of 304 (SC_NOT_MODIFIED) means that the cached version is up-to-date and the client should use it. Otherwise, the server should return the requested document with the normal (200) status code. Servlets normally should not set this status code directly. Instead, they should implement the getLastModified method and let the default service method handle conditional requests based upon this modification date. For an example, see Section 2.8 of Core Servlets and JavaServer Pages.

307 (Temporary Redirect)
The rules for how a browser should handle a 307 status are identical to those for 302. The 307 value was added to HTTP 1.1 since many browsers erroneously follow the redirection on a 302 response even if the original message is a POST. Browsers are supposed to follow the redirection of a POST request only when they receive a 303 response status. This new status is intended to be unambiguously clear: follow redirected GET and POST requests in the case of 303 responses; follow redirected GET but not POST requests in the case of 307 responses. Note: For some reason there is no constant in HttpServletResponse corresponding to this status code, so you have to use 307 explicitly. This status code is new in HTTP 1.1.

400 (Bad Request)
A 400 (SC_BAD_REQUEST) status indicates bad syntax in the client request.

401 (Unauthorized)
A value of 401 (SC_UNAUTHORIZED) signifies that the client tried to access a password-protected page without proper identifying information in the Authorization header. The response must include a WWW-Authenticate header.

403 (Forbidden)
A status code of 403 (SC_FORBIDDEN) means that the server refuses to supply the resource, regardless of authorization. This status is often the result of bad file or directory permissions on the server.

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
404 (Not Found)

The infamous 404 (SC_NOT_FOUND) status tells the client that no resource could be found at that address. This value is the standard “no such page” response. It is such a common and useful response that there is a special method for it in the HttpServletResponse class: sendError("message"). The advantage of sendError over setStatus is that, with sendError, the server automatically generates an error page showing the error message. 404 errors need not merely say “Sorry, the page cannot be found.” Instead, they can give information on why the page couldn’t be found or supply search boxes or alternative places to look. The sites at www.microsoft.com and www.ibm.com have particularly good examples of useful error pages. In fact, there is an entire site dedicated to the good, the bad, the ugly, and the bizarre in 404 error messages: http://www.plinko.net/404/. I find http://www.plinko.net/404/category.asp?Category=Funny particularly amusing.

Unfortunately, however, the default behavior of Internet Explorer 5 is to ignore the error page you send back and to display its own, even though doing so contradicts the HTTP specification. To turn off this setting, you can go to the Tools menu, select Internet Options, choose the Advanced tab, and make sure “Show friendly HTTP error messages” box is not checked. Unfortunately, however, few users are aware of this setting, so this “feature” prevents most users of Internet Explorer version 5 from seeing any informative messages you return. Other major browsers and version 4 of Internet Explorer properly display server-generated error pages.

Core Warning

By default, Internet Explorer version 5 improperly ignores server-generated error pages.

To make matters worse, some versions of Tomcat 3 fail to properly handle strings that are passed to sendError. So, if you are using Tomcat 3, you may need to generate 404 error messages by hand. Fortunately, it is relatively uncommon for individual servlets to build their own 404 error pages. A more common approach is to set up error pages for each Web application; see Section 5.8 (Designating Pages to Handle Errors) for details. Tomcat correctly handles these pages.
2.7 The Server Response: HTTP Status Codes

Core Warning

Some versions of Tomcat 3.x fail to properly display strings that are supplied to sendError.

405 (Method Not Allowed)
A 405 (SC_METHOD_NOT_ALLOWED) value indicates that the request method (GET, POST, HEAD, PUT, DELETE, etc.) was not allowed for this particular resource. This status code is new in HTTP 1.1.

415 (Unsupported Media Type)
A value of 415 (SC_UNSUPPORTED_MEDIA_TYPE) means that the request had an attached document of a type the server doesn’t know how to handle. This status code is new in HTTP 1.1.

417 (Expectation Failed)
If the server receives an Expect request header with a value of 100-continue, it means that the client is asking if it can send an attached document in a follow-up request. In such a case, the server should either respond with this status (417) to tell the browser it won’t accept the document or use 100 (SC_CONTINUE) to tell the client to go ahead. This status code is new in HTTP 1.1.

500 (Internal Server Error)
500 (SC_INTERNAL_SERVER_ERROR) is the generic “server is confused” status code. It often results from CGI programs or (heaven forbid!) servlets that crash or return improperly formatted headers.

501 (Not Implemented)
The 501 (SC_NOT_IMPLEMENTED) status notifies the client that the server doesn’t support the functionality to fulfill the request. It is used, for example, when the client issues a command like PUT that the server doesn’t support.

503 (Service Unavailable)
A status code of 503 (SC_SERVICE_UNAVAILABLE) signifies that the server cannot respond because of maintenance or overloading. For example, a servlet might return this header if some thread or database connection pool is currently full. The server can supply a Retry-After header to tell the client when to try again.

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
505 (HTTP Version Not Supported)

The 505 (SC_HTTP_VERSION_NOT_SUPPORTED) code means that the server doesn’t support the version of HTTP named in the request line. This status code is new in HTTP 1.1.

A Front End to Various Search Engines

Listing 2.12 presents an example that makes use of the two most common status codes other than 200 (OK): 302 (Found) and 404 (Not Found). The 302 code is set by the shorthand sendRedirect method of HttpServletResponse, and 404 is specified by sendError.

In this application, an HTML form (see Figure 2–10 and the source code in Listing 2.14) first displays a page that lets the user specify a search string, the number of results to show per page, and the search engine to use. When the form is submitted, the servlet extracts those three parameters, constructs a URL with the parameters embedded in a way appropriate to the search engine selected (see the SearchSpec class of Listing 2.13), and redirects the user to that URL (see Figure 2–11). If the user fails to choose a search engine or specify search terms, an error page informs the client of this fact (but see warnings under the 404 status code in the previous subsection).

Listing 2.12 SearchEngines.java

```java
package moreServlets;

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

/** Servlet that takes a search string, number of results per
 * page, and a search engine name, sending the query to
 * that search engine. Illustrates manipulating
 * the response status line. It sends a 302 response
 * (via sendRedirect) if it gets a known search engine,
 * and sends a 404 response (via sendError) otherwise. */

public class SearchEngines extends HttpServlet {
    public void doGet(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {
        String searchString = request.getParameter("searchString");
    }
}
```

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
if ((searchString == null) ||
    (searchString.length() == 0)) {
    reportProblem(response, "Missing search string.");
    return;
}
// The URLEncoder changes spaces to "+" signs and other
// non-alphanumeric characters to "%XY", where XY is the
// hex value of the ASCII (or ISO Latin-1) character.
// Browsers always URL-encode form values, so the
// getParameter method decodes automatically. But since
// we're just passing this on to another server, we need to
// re-encode it.
searchString = URLEncoder.encode(searchString);
String numResults = request.getParameter("numResults");
if ((numResults == null) ||
    (numResults.equals("0") ||
    (numResults.length() == 0)) {  
    numResults = "10";
}
String searchEngine =
    request.getParameter("searchEngine");
if (searchEngine == null) {
    reportProblem(response, "Missing search engine name.");
    return;
}
SearchSpec[] commonSpecs = SearchSpec.getCommonSpecs();
for (int i=0; i<commonSpecs.length; i++) {
    SearchSpec searchSpec = commonSpecs[i];
    if (searchSpec.getName().equals(searchEngine)) {
        String url =
            searchSpec.makeURL(searchString, numResults);
        response.sendRedirect(url);
        return;
    }
}
reportProblem(response, "Unrecognized search engine.");
}

private void reportProblem(HttpServletResponse response,
    String message)
    throws IOException {
    response.sendError(response.SC_NOT_FOUND,
        "<H2>" + message + "</H2>");
}
Chapter 2: A Fast Introduction to Basic Servlet Programming

**Listing 2.12** SearchEngines.java (continued)

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

**Listing 2.13** SearchSpec.java

```java
package moreservlets;

/** Small class that encapsulates how to construct a
 *  search string for a particular search engine. *
 */
public class SearchSpec {
    private String name, baseURL, numResultsSuffix;

    private static SearchSpec[] commonSpecs =
    { new SearchSpec
        ("google",
         "http://www.google.com/search?q=", "&num=")
        , new SearchSpec
        ("altavista",
         "http://www.altavista.com/sites/search/web?q=", "&nbq=")
        , new SearchSpec
        ("lycos",
         "http://lycospro.lycos.com/cgi-bin/" +
         "pursuit?query=", "&maxhits=")
        , new SearchSpec
        ("hotbot",
         "http://www.hotbot.com/?MT=", "&DC=")
    };

    public SearchSpec(String name,
                       String baseURL,
                       String numResultsSuffix) {
```
Listing 2.13  

`SearchSpec.java (continued)`

```java
this.name = name;
this.baseURL = baseURL;
this.numResultsSuffix = numResultsSuffix;
}

public String makeURL(String searchString,
                        String numResults) {
    return(baseURL + searchString +
            numResultsSuffix + numResults);
}

public String getName() {
    return(name);
}

public static SearchSpec[] getCommonSpecs() {
    return(commonSpecs);
}
```

Figure 2–10  
Front end to the `SearchEngines` servlet. See Listing 2.14 for the HTML source code.

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/.
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
2.8 The Server Response: HTTP Response Headers

As discussed in the previous section, a response from a Web server normally consists of a status line, one or more response headers (one of which must be Content-Type), a blank line, and the document. To get the most out of your servlets, you need to

Listing 2.14 SearchEngines.html

```html
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
  <title>Searching the Web</title>
</head>
<body bgcolor="#FDF5E6">
  <h1 align="center">Searching the Web</h1>

  <form action="/servlet/moreservlets.SearchEngines" method="get">
    <center>
      Search String: <input type="text" name="searchString"><br>
      Results to Show Per Page: <input type="text" name="numResults" value="10" size="3"><br>
      <input type="radio" name="searchEngine" value="google"> Google | <input type="radio" name="searchEngine" value="altavista"> AltaVista | <input type="radio" name="searchEngine" value="lycos"> Lycos | <input type="radio" name="searchEngine" value="hotbot"> HotBot<br>
      <input type="submit" value="Search"> </center>
  </form>
</body>
</html>
```
know how to use the status line and response headers effectively, not just how to generate the document.

Setting the HTTP response headers often goes hand in hand with setting the status codes in the status line, as discussed in the previous section. For example, all the “document moved” status codes (300 through 307) have an accompanying Location header, and a 401 (Unauthorized) code always includes an accompanying WWW-Authenticate header. However, specifying headers can also play a useful role even when no unusual status code is set. Response headers can be used to specify cookies, to supply the page modification date (for client-side caching), to instruct the browser to reload the page after a designated interval, to give the file size so that persistent HTTP connections can be used, to designate the type of document being generated, and to perform many other tasks. This section gives a brief summary of the handling of response headers. See Chapter 7 of Core Servlets and JavaServer Pages (available in PDF at http://www.moreservlets.com) for more details and examples.

### Setting Response Headers from Servlets

The most general way to specify headers is to use the setHeader method of HttpServletResponse. This method takes two strings: the header name and the header value. As with setting status codes, you must specify headers before returning the actual document.

In addition to the general-purpose setHeader method, HttpServletResponse also has two specialized methods to set headers that contain dates and integers:

- **setDateHeader(String header, long milliseconds)**
  This method saves you the trouble of translating a Java date in milliseconds since 1970 (as returned by System.currentTimeMillis, Date.getTime, or Calendar.getTimeInMillis) into a GMT time string.

- **setIntHeader(String header, int headerValue)**
  This method spares you the minor inconvenience of converting an int to a String before inserting it into a header.

HTTP allows multiple occurrences of the same header name, and you sometimes want to add a new header rather than replace any existing header with the same name. For example, it is quite common to have multiple Accept and Set-Cookie headers that specify different supported MIME types and different cookies, respectively. With servlets version 2.1, setHeader, setDateHeader, and setIntHeader always add new headers, so there is no way to “unset” headers that were set earlier (e.g., by an inherited method). With servlets versions 2.2 and 2.3, setHeader, setDateHeader, and setIntHeader replace any existing headers of the
same name, whereas addHeader, addDateHeader, and addIntHeader add a header regardless of whether a header of that name already exists. If it matters to you whether a specific header has already been set, use containsHeader to check.

Finally, HttpServletResponse also supplies a number of convenience methods for specifying common headers. These methods are summarized as follows.

- **setContentType**
  This method sets the Content-Type header and is used by the majority of servlets.

- **setContentLength**
  This method sets the Content-Length header, which is useful if the browser supports persistent (keep-alive) HTTP connections.

- **addCookie**
  This method inserts a cookie into the Set-Cookie header. There is no corresponding setCookie method, since it is normal to have multiple Set-Cookie lines. See Section 2.9 (Cookies) for a discussion of cookies.

- **sendRedirect**
  As discussed in the previous section, the sendRedirect method sets the Location header as well as setting the status code to 302. See Listing 2.12 for an example.

### Understanding HTTP 1.1 Response Headers

Following is a summary of the most useful HTTP 1.1 response headers. A good understanding of these headers can increase the effectiveness of your servlets, so you should at least skim the descriptions to see what options are at your disposal. You can come back for details when you are ready to use the capabilities.

These headers are a superset of those permitted in HTTP 1.0. The official HTTP 1.1 specification is given in RFC 2616. The RFCs are online in various places; your best bet is to start at [http://www.rfc-editor.org/](http://www.rfc-editor.org/) to get a current list of the archive sites. Header names are not case sensitive but are traditionally written with the first letter of each word capitalized.

Be cautious in writing servlets whose behavior depends on response headers that are only available in HTTP 1.1, especially if your servlet needs to run on the WWW “at large” rather than on an intranet—many older browsers support only HTTP 1.0. It is best to explicitly check the HTTP version with request.getRequestProtocol before using new headers.

- **Allow**
  The Allow header specifies the request methods (GET, POST, etc.) that the server supports. It is required for 405 (Method Not Allowed) responses.
The default `service` method of servlets automatically generates this header for `OPTIONS` requests.

**Cache-Control**

This useful header tells the browser or other client the circumstances in which the response document can safely be cached. It has the following possible values:

- **public.** Document is cacheable, even if normal rules (e.g., for password-protected pages) indicate that it shouldn’t be.
- **private.** Document is for a single user and can only be stored in private (nonshared) caches.
- **no-cache.** Document should never be cached (i.e., used to satisfy a later request). The server can also specify “no-cache="header1,header2,...,headerN"” to indicate the headers that should be omitted if a cached response is later used. Browsers normally do not cache documents that were retrieved by requests that include form data. However, if a servlet generates different content for different requests even when the requests contain no form data, it is critical to tell the browser not to cache the response. Since older browsers use the `Pragma` header for this purpose, the typical servlet approach is to set both headers, as in the following example.

```java
response.setHeader("Cache-Control", "no-cache");
response.setHeader("Pragma", "no-cache");
```

- **no-store.** Document should never be cached and should not even be stored in a temporary location on disk. This header is intended to prevent inadvertent copies of sensitive information.
- **must-revalidate.** Client must revalidate document with original server (not just intermediate proxies) each time it is used.
- **proxy-revalidate.** This is the same as `must-revalidate`, except that it applies only to shared caches.
- **max-age=xxx.** Document should be considered stale after `xxx` seconds. This is a convenient alternative to the `Expires` header but only works with HTTP 1.1 clients. If both `max-age` and `Expires` are present in the response, the `max-age` value takes precedence.
- **s-max-age=xxx.** Shared caches should consider the document stale after `xxx` seconds.

The `Cache-Control` header is new in HTTP 1.1.
Connection
A value of close for this response header instructs the browser not to use persistent HTTP connections. Technically, persistent connections are the default when the client supports HTTP 1.1 and does not specify a Connection: close request header (or when an HTTP 1.0 client specifies Connection: keep-alive). However, since persistent connections require a Content-Length response header, there is no reason for a servlet to explicitly use the Connection header. Just omit the Content-Length header if you aren’t using persistent connections.

Content-Encoding
This header indicates the way in which the page was encoded during transmission. The browser should reverse the encoding before deciding what to do with the document. Compressing the document with gzip can result in huge savings in transmission time; for an example, see Section 9.5.

Content-Language
The Content-Language header signifies the language in which the document is written. The value of the header should be one of the standard language codes such as en, en-us, da, etc. See RFC 1766 for details on language codes (you can access RFCs online at one of the archive sites listed at http://www.rfc-editor.org/).

Content-Length
This header indicates the number of bytes in the response. This information is needed only if the browser is using a persistent (keep-alive) HTTP connection. See the Connection header for determining when the browser supports persistent connections. If you want your servlet to take advantage of persistent connections when the browser supports it, your servlet should write the document into a ByteArrayOutputStream, look up its size when done, put that into the Content-Length field with response.setContentType. MIME types are of the form maintype/subtype for officially registered types and of the form maintype/x-subtype for

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
unregistered types. Most servlets specify `text/html`; they can, however, specify other types instead.

In addition to a basic MIME type, the `Content-Type` header can also designate a specific character encoding. If this is not specified, the default is `ISO-8859_1` (Latin). For example, the following instructs the browser to interpret the document as HTML in the `Shift_JIS` (standard Japanese) character set.

```java
response.setContentType("text/html; charset=Shift_JIS");
```

Table 2.1 lists some of the most common MIME types used by servlets. RFC 1521 and RFC 1522 list more of the common MIME types (again, see [http://www.rfc-editor.org/](http://www.rfc-editor.org/) for a list of RFC archive sites). However, new MIME types are registered all the time, so a dynamic list is a better place to look. The officially registered types are listed at [http://www.isi.edu/in-notes/iana/assignments/media-types/media-types](http://www.isi.edu/in-notes/iana/assignments/media-types/media-types). For common unregistered types, [http://www.ltsw.se/knbase/internet/mime.htp](http://www.ltsw.se/knbase/internet/mime.htp) is a good source.

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>application/msword</td>
<td>Microsoft Word document</td>
</tr>
<tr>
<td>application/octet-stream</td>
<td>Unrecognized or binary data</td>
</tr>
<tr>
<td>application/pdf</td>
<td>Acrobat (.pdf) file</td>
</tr>
<tr>
<td>application/postscript</td>
<td>PostScript file</td>
</tr>
<tr>
<td>application/vnd.lotus-notes</td>
<td>Lotus Notes file</td>
</tr>
<tr>
<td>application/vnd.ms-excel</td>
<td>Excel spreadsheet</td>
</tr>
<tr>
<td>application/vnd.ms-powerpoint</td>
<td>PowerPoint presentation</td>
</tr>
<tr>
<td>application/x-gzip</td>
<td>Gzip archive</td>
</tr>
<tr>
<td>application/x-java-archive</td>
<td>JAR file</td>
</tr>
<tr>
<td>application/x-java-serialized-object</td>
<td>Serialized Java object</td>
</tr>
<tr>
<td>application/x-java-vm</td>
<td>Java bytecode (.class) file</td>
</tr>
<tr>
<td>application/zip</td>
<td>Zip archive</td>
</tr>
<tr>
<td>audio/basic</td>
<td>Sound file in .au or .snd format</td>
</tr>
</tbody>
</table>

J2EE training from the author: [http://courses.coreservlets.com/](http://courses.coreservlets.com/)
Expires
This header stipulates the time at which the content should be considered out-of-date and thus no longer be cached. A servlet might use this for a document that changes relatively frequently, to prevent the browser from displaying a stale cached value. Furthermore, since some older browsers support Pragma unreliably (and Cache-Control not at all), an Expires header with a date in the past is often used to prevent browser caching.

For example, the following would instruct the browser not to cache the document for more than 10 minutes.

```java
long currentTime = System.currentTimeMillis();
long tenMinutes = 10*60*1000; // In milliseconds
response.setDateHeader("Expires", currentTime + tenMinutes);
```

Also see the max-age value of the Cache-Control header.

---

**Table 2.1  Common MIME Types (continued)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>audio/midi</td>
<td>MIDI sound file</td>
</tr>
<tr>
<td>audio/x-aiff</td>
<td>AIFF sound file</td>
</tr>
<tr>
<td>audio/x-wav</td>
<td>Microsoft Windows sound file</td>
</tr>
<tr>
<td>image/gif</td>
<td>GIF image</td>
</tr>
<tr>
<td>image/jpeg</td>
<td>JPEG image</td>
</tr>
<tr>
<td>image/png</td>
<td>PNG image</td>
</tr>
<tr>
<td>image/tiff</td>
<td>TIFF image</td>
</tr>
<tr>
<td>image/x-xbitmap</td>
<td>X Windows bitmap image</td>
</tr>
<tr>
<td>text/css</td>
<td>HTML cascading style sheet</td>
</tr>
<tr>
<td>text/html</td>
<td>HTML document</td>
</tr>
<tr>
<td>text/plain</td>
<td>Plain text</td>
</tr>
<tr>
<td>text/xml</td>
<td>XML</td>
</tr>
<tr>
<td>video/mpeg</td>
<td>MPEG video clip</td>
</tr>
<tr>
<td>video/quicktime</td>
<td>QuickTime video clip</td>
</tr>
</tbody>
</table>

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
Last-Modified
This very useful header indicates when the document was last changed. The client can then cache the document and supply a date by an If-Modified-Since request header in later requests. This request is treated as a conditional GET, with the document being returned only if the Last-Modified date is later than the one specified for If-Modified-Since. Otherwise, a 304 (Not Modified) status line is returned, and the client uses the cached document. If you set this header explicitly, use the setDateHeader method to save yourself the bother of formatting GMT date strings. However, in most cases you simply implement the getLastModified method (see Core Servlets and JavaServer Pages Section 2.8) and let the standard service method handle If-Modified-Since requests.

Location
This header, which should be included with all responses that have a status code in the 300s, notifies the browser of the document address. The browser automatically reconnects to this location and retrieves the new document. This header is usually set indirectly, along with a 302 status code, by the sendRedirect method of HttpServletResponse. An example is given in the previous section (Listing 2.12).

Pragma
Supplying this header with a value of no-cache instructs HTTP 1.0 clients not to cache the document. However, support for this header was inconsistent with HTTP 1.0 browsers, so Expires with a date in the past is often used instead. In HTTP 1.1, Cache-Control: no-cache is a more reliable replacement.

Refresh
This header indicates how soon (in seconds) the browser should ask for an updated page. For example, to tell the browser to ask for a new copy in 30 seconds, you would specify a value of 30 with

```java
response.setIntHeader("Refresh", 30)
```

Note that Refresh does not stipulate continual updates; it just specifies when the next update should be. So, you have to continue to supply Refresh in all subsequent responses. This header is extremely useful because it lets servlets return partial results quickly while still letting the client see the complete results at a later time. For an example, see Section 7.3 of Core Servlets and JavaServer Pages (in PDF at http://www.moreservlets.com).
Instead of having the browser just reload the current page, you can specify the page to load. You do this by supplying a semicolon and a URL after the refresh time. For example, to tell the browser to go to http://host/path after 5 seconds, you would do the following.

    response.setHeader("Refresh", "5; URL=http://host/path/")

This setting is useful for “splash screens,” where an introductory image or message is displayed briefly before the real page is loaded.

Note that this header is commonly set indirectly by putting

    <META HTTP-EQUIV="Refresh" CONTENT="5; URL=http://host/path/">

in the HEAD section of the HTML page, rather than as an explicit header from the server. That usage came about because automatic reloading or forwarding is something often desired by authors of static HTML pages. For servlets, however, setting the header directly is easier and clearer.

This header is not officially part of HTTP 1.1 but is an extension supported by both Netscape and Internet Explorer.

**Retry-After**

This header can be used in conjunction with a 503 (Service Unavailable) response to tell the client how soon it can repeat its request.

**Set-Cookie**

The Set-Cookie header specifies a cookie associated with the page. Each cookie requires a separate Set-Cookie header. Servlets should not use `response.setHeader("Set-Cookie", ...)` but instead should use the special-purpose `addCookie` method of `HttpServletResponse`. For details, see Section 2.9 (Cookies). Technically, `Set-Cookie` is not part of HTTP 1.1. It was originally a Netscape extension but is now widely supported, including in both Netscape and Internet Explorer.

**WWW-Authenticate**

This header is always included with a 401 (Unauthorized) status code. It tells the browser what authorization type (BASIC or DIGEST) and realm the client should supply in its Authorization header. See Chapters 7 and 8 for a discussion of the various security mechanisms available to servlets.
2.9 Cookies

Cookies are small bits of textual information that a Web server sends to a browser and that the browser later returns unchanged when visiting the same Web site or domain. By letting the server read information it sent the client previously, the site can provide visitors with a number of conveniences such as presenting the site the way the visitor previously customized it or letting identifiable visitors in without their having to reenter a password.

This section discusses how to explicitly set and read cookies from within servlets, and the next section shows how to use the servlet session tracking API (which can use cookies behind the scenes) to keep track of users as they move around to different pages within your site.

Benefits of Cookies

There are four typical ways in which cookies can add value to your site.

Identifying a User During an E-commerce Session

Many online stores use a "shopping cart" metaphor in which the user selects an item, adds it to his shopping cart, then continues shopping. Since the HTTP connection is usually closed after each page is sent, when the user selects a new item to add to the cart, how does the store know that it is the same user who put the previous item in the cart? Persistent (keep-alive) HTTP connections do not solve this problem, since persistent connections generally apply only to requests made very close together in time, as when a browser asks for the images associated with a Web page. Besides, many servers and browsers lack support for persistent connections. Cookies, however, can solve this problem. In fact, this capability is so useful that servlets have an API specifically for session tracking, and servlet authors don’t need to manipulate cookies directly to take advantage of it. Session tracking is discussed in Section 2.10.

Avoiding Username and Password

Many large sites require you to register to use their services, but it is inconvenient to remember and enter the username and password each time you visit. Cookies are a good alternative for low-security sites. When a user registers, a cookie containing a unique user ID is sent to him. When the client reconnects at a later date, the user ID is returned automatically; the server looks it up, determines it belongs to a registered user, and permits access without an explicit username and password. The site might also store the user’s address, credit card number, and so forth in a database and use the user ID from the cookie as a key to retrieve the data. This approach prevents the user from having to reenter the data each time.

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
Customizing a Site

Many “portal” sites let you customize the look of the main page. They might let you pick which weather report you want to see, what stock and sports results you care about, how search results should be displayed, and so forth. Since it would be inconvenient for you to have to set up your page each time you visit their site, they use cookies to remember what you wanted. For simple settings, the site could accomplish this customization by storing the page settings directly in the cookies. For more complex customization, however, the site just sends the client a unique identifier and keeps a server-side database that associates identifiers with page settings.

Focusing Advertising

Most advertiser-funded Web sites charge their advertisers much more for displaying “directed” ads than “random” ads. Advertisers are generally willing to pay much more to have their ads shown to people that are known to have some interest in the general product category. For example, if you go to a search engine and do a search on “Java Servlets,” the search site can charge an advertiser much more for showing you an ad for a servlet development environment than for an ad for an online travel agent specializing in Indonesia. On the other hand, if the search had been for “Java Hotels,” the situation would be reversed. Without cookies, the sites have to show a random ad when you first arrive and haven’t yet performed a search, as well as when you search on something that doesn’t match any ad categories. With cookies, they can identify your interests by remembering your previous searches.

Some Problems with Cookies

Providing convenience to the user and added value to the site owner is the purpose behind cookies. And despite much misinformation, cookies are not a serious security threat. Cookies are never interpreted or executed in any way and thus cannot be used to insert viruses or attack your system. Furthermore, since browsers generally only accept 20 cookies per site and 300 cookies total, and since browsers can limit each cookie to 4 kilobytes, cookies cannot be used to fill up someone’s disk or launch other denial-of-service attacks.

However, even though cookies don’t present a serious security threat, they can present a significant threat to privacy. First, some people don’t like the fact that search engines can remember that they’re the user who usually does searches on certain topics. For example, they might search for job openings or sensitive health data and don’t want some banner ad tipping off their coworkers next time they do a search. Even worse, two sites can share data on a user by each loading small images off the same third-party site, where that third party uses cookies and shares the data with both original sites. The doubleclick.net service is the prime example of this technique. (Netscape, however, provides a nice feature that lets you refuse cookies.
from sites other than that to which you connected, but without disabling cookies altogether.) This trick of associating cookies with images can even be exploited through e-mail if you use an HTML-enabled e-mail reader that “supports” cookies and is associated with a browser. Thus, people could send you e-mail that loads images, attach cookies to those images, then identify you (e-mail address and all) if you subsequently visit their Web site. Boo.

A second privacy problem occurs when sites rely on cookies for overly sensitive data. For example, some of the big online bookstores use cookies to remember users and let you order without reentering much of your personal information. This is not a particular problem since they don’t actually display the full credit card number and only let you send books to an address that was specified when you did enter the credit card in full or use the username and password. As a result, someone using your computer (or stealing your cookie file) could do no more harm than sending a big book order to your address, where the order could be refused. However, other companies might not be so careful, and an attacker who gained access to someone’s computer or cookie file could get online access to valuable personal information. Even worse, incompetent sites might embed credit card or other sensitive information directly in the cookies themselves, rather than using innocuous identifiers that are only linked to real users on the server. This is dangerous, since most users don’t view leaving their computer unattended in their office as being tantamount to leaving their credit card sitting on their desk.

The point of this discussion is twofold. First, due to real and perceived privacy problems, some users turn off cookies. So, even when you use cookies to give added value to a site, your site shouldn’t depend on them. Second, as the author of servlets that use cookies, you should be careful not to use cookies for particularly sensitive information, since this would open users up to risks if somebody accessed their computer or cookie files.

**The Servlet Cookie API**

To send cookies to the client, a servlet should create one or more cookies with designated names and values with new Cookie(name, value), set any optional attributes with cookie.setXxx (readable later by cookie.getXxx), and insert the cookies into the response headers with response.addCookie(cookie). To read incoming cookies, a servlet should call request.getCookies, which returns an array of Cookie objects corresponding to the cookies the browser has associated with your site (null if there are no cookies in the request). In most cases, the servlet loops down this array until it finds the one whose name (getName) matches the name it had in mind, then calls getValue on that Cookie to see the value associated with that name. Each of these topics is discussed in more detail in the following sections.
Creating Cookies

You create a cookie by calling the Cookie constructor, which takes two strings: the cookie name and the cookie value. Neither the name nor the value should contain white space or any of the following characters:

\[
[ ] () = , / ? @ : ;
\]

If you want the browser to store the cookie on disk instead of just keeping it in memory, use setMaxAge to specify how long (in seconds) the cookie should be valid.

Placing Cookies in the Response Headers

The cookie is inserted into a Set-Cookie HTTP response header by means of the addCookie method of HttpServletResponse. The method is called addCookie, not setCookie, because any previously specified Set-Cookie headers are left alone and a new header is set. Here's an example:

```java
Cookie userCookie = new Cookie("user", "uid1234");
userCookie.setMaxAge(60*60*24*365); // Store cookie for 1 year
response.addCookie(userCookie);
```

Reading Cookies from the Client

To send cookies to the client, you create a Cookie, then use addCookie to send a Set-Cookie HTTP response header. To read the cookies that come back from the client, you call getCookies on the HttpServletRequest. This call returns an array of Cookie objects corresponding to the values that came in on the Cookie HTTP request header. If the request contains no cookies, getCookies should return null. However, Tomcat 3.x returns a zero-length array instead.

Core Warning

In Tomcat 3.x, calls to request.getCookies return a zero-length array instead of null when there are no cookies in the request. Tomcat 4 and most commercial servers properly return null.

Once you have this array, you typically loop down it, calling getName on each Cookie until you find one matching the name you have in mind. You then call getValue on the matching Cookie and finish with some processing specific to the resultant value. This is such a common process that, at the end of this section, I present two utilities that simplify retrieving a cookie or cookie value that matches a designated cookie name.

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
Using Cookie Attributes

Before adding the cookie to the outgoing headers, you can set various characteristics of the cookie by using one of the following `setXxx` methods, where `Xxx` is the name of the attribute you want to specify. Each `setXxx` method has a corresponding `getXxx` method to retrieve the attribute value. Except for name and value, the cookie attributes apply only to outgoing cookies from the server to the client; they aren't set on cookies that come from the browser to the server. So, don't expect these attributes to be available in the cookies you get by means of `request.getCookies`.

```
public String getComment()
public void setComment(String comment)
```

These methods look up or specify a comment associated with the cookie. With version 0 cookies (see the upcoming entry on `getVersion` and `setVersion`), the comment is used purely for informational purposes on the server; it is not sent to the client.

```
public String getDomain()
public void setDomain(String domainPattern)
```

These methods get or set the domain to which the cookie applies. Normally, the browser returns cookies only to the same hostname that sent them. You can use `setDomain` method to instruct the browser to return them to other hosts within the same domain. To prevent servers from setting cookies that apply to hosts outside their domain, the specified domain must meet the following two requirements: It must start with a dot (e.g., `.prenhall.com`); it must contain two dots for noncountry domains like `.com`, `.edu`, and `.gov`; and it must contain three dots for country domains like `.co.uk` and `.edu.es`. For instance, cookies sent from a servlet at `bali.vacations.com` would not normally get returned by the browser to pages at `mexico.vacations.com`. If the site wanted this to happen, the servlets could specify `cookie.setDomain(".vacations.com")`.

```
public int getMaxAge()
public void setMaxAge(int lifetime)
```

These methods tell how much time (in seconds) should elapse before the cookie expires. A negative value, which is the default, indicates that the cookie will last only for the current session (i.e., until the user quits the browser) and will not be stored on disk. See the `LongLivedCookie` class (Listing 2.18), which defines a subclass of `Cookie` with a maximum age automatically set one year in the future. Specifying a value of 0 instructs the browser to delete the cookie.

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
2.9 Cookies

public String getName()
public void setName(String cookieName)
This pair of methods gets or sets the name of the cookie. The name and the value are the two pieces you virtually always care about. However, since the name is supplied to the Cookie constructor, you rarely need to call setName. On the other hand, getName is used on almost every cookie received on the server. Since the getCookies method of HttpServletRequest returns an array of Cookie objects, a common practice is to loop down the array, calling getName until you have a particular name, then to check the value with getValue. For an encapsulation of this process, see the getCookieValue method shown in Listing 2.17.

public String getPath()
public void setPath(String path)
These methods get or set the path to which the cookie applies. If you don’t specify a path, the browser returns the cookie only to URLs in or below the directory containing the page that sent the cookie. For example, if the server sent the cookie from http://ecommerce.site.com/toys/specials.html, the browser would send the cookie back when connecting to http://ecommerce.site.com/toys/bikes/beginners.html, but not to http://ecommerce.site.com/cds/classical.html. The setPath method can specify something more general. For example, someCookie.setPath("/") specifies that all pages on the server should receive the cookie. The path specified must include the current page; that is, you may specify a more general path than the default, but not a more specific one. So, for example, a servlet at http://host/store/cust-service/request could specify a path of /store/ (since /store/ includes /store/cust-service/) but not a path of /store/cust-service/returns/ (since this directory does not include /store/cust-service/).

public boolean getSecure()
public void setSecure(boolean secureFlag)
This pair of methods gets or sets the boolean value indicating whether the cookie should only be sent over encrypted (i.e., SSL) connections. The default is false; the cookie should apply to all connections.

public String getValue()
public void setValue(String cookieValue)
The getValue method looks up the value associated with the cookie; the setValue method specifies it. Again, the name and the value are the two parts of a cookie that you almost always care about, although in a few cases, a name
is used as a boolean flag and its value is ignored (i.e., the existence of a cookie with the designated name is all that matters).

```java
public int getVersion()
public void setVersion(int version)
```

These methods get and set the cookie protocol version the cookie complies with. Version 0, the default, follows the original Netscape specification (http://www.netscape.com/newsref/std/cookie_spec.html). Version 1, not yet widely supported, adheres to RFC 2109 (retrieve RFCs from the archive sites listed at http://www.rfc-editor.org/).

Examples of Setting and Reading Cookies

Listing 2.15 and Figure 2–13 show the SetCookies servlet, a servlet that sets six cookies. Three have the default expiration date, meaning that they should apply only until the user next restarts the browser. The other three use `setMaxAge` to stipulate that they should apply for the next hour, regardless of whether the user restarts the browser or reboots the computer to initiate a new browsing session.

Listing 2.16 shows a servlet that creates a table of all the cookies sent to it in the request. Figure 2–14 shows this servlet immediately after the SetCookies servlet is visited. Figure 2–15 shows it within an hour of when SetCookies is visited but when the browser has been closed and restarted. Figure 2–16 shows it within an hour of when SetCookies is visited but when the browser has been closed and restarted.

Listing 2.15 SetCookies.java

```java
package moreservlets;

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

/** Sets six cookies: three that apply only to the current session (regardless of how long that session lasts) and three that persist for an hour (regardless of whether the browser is restarted).
*/
```
public class SetCookies extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        for(int i=0; i<3; i++) {
            // Default maxAge is -1, indicating cookie applies only to current browsing session.
            Cookie cookie = new Cookie("Session-Cookie-" + i,
                "Cookie-Value-S" + i);
            response.addCookie(cookie);
            cookie = new Cookie("Persistent-Cookie-" + i,
                "Cookie-Value-P" + i);
            // Cookie is valid for an hour, regardless of whether user quits browser, reboots computer, or whatever.
            cookie.setMaxAge(3600);
            response.addCookie(cookie);
        }
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Setting Cookies";
        out.println((ServletUtilities.headWithTitle(title) +
            "<BODY BGCOLOR="#FDF5E6">
            <H1 ALIGN="CENTER">" + title + "</H1>"
            + "There are six cookies associated with this page." +
            "To see them, visit the
            <A HREF="/servlet/moreservlets.ShowCookies">
            <CODE>ShowCookies</CODE> servlet</A>.
            "
            "Three of the cookies are associated only with the current session, while three are persistent."
            "Quit the browser, restart, and return to the
            <CODE>ShowCookies</CODE> servlet to verify that the three long-lived ones persist across sessions.";
            "</BODY></HTML>");
    }
}
Chapter 2: A Fast Introduction to Basic Servlet Programming

100

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/

Figure 2–13 Result of SetCookies servlet.

Listing 2.16 ShowCookies.java

package morebservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
/** Creates a table of the cookies associated with
*  the current page.
*/
public class ShowCookies extends HttpServlet {
    public void doGet(HttpServletRequest request,
                        HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Active Cookies";
}
2.9 Cookies

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/.
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/

Figure 2–14  Result of visiting the ShowCookies servlet within an hour of visiting SetCookies (same browser session).

Listing 2.16  ShowCookies.java (continued)

```java
out.println(ServletUtilities.headWithTitle(title) +
"<BODY BGCOLOR="#FDF5E6">
" + title + "</H1>
" +
"<TABLE BORDER=1 ALIGN="CENTER">
" +
"<TR BGCOLOR="#FFAD00">
" +
"<TH>Cookie Name" +
"<TH>Cookie Value";

Cookie[] cookies = request.getCookies();
if (cookies == null) {
  out.println("<TR><TH COLSPAN=2>No cookies");
} else {
  Cookie cookie;
  for(int i=0; i<cookies.length; i++) {
    cookie = cookies[i];
    out.println("<TR>
" +
      "  <TD>" + cookie.getName() + "\n" +
      "  <TD>" + cookie.getValue();
  }
}  
out.println("</TABLE></BODY></HTML>";
```
Figure 2–15  Result of visiting the ShowCookies servlet within an hour of visiting SetCookies (different browser session).

Figure 2–16  Result of visiting the ShowCookies servlet more than an hour after visiting SetCookies (different browser session).

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
### Basic Cookie Utilities

This section presents some simple but useful utilities for dealing with cookies.

#### Finding Cookies with Specified Names

Listing 2.17 shows a section of `ServletUtilities.java` that simplifies the retrieval of a cookie or cookie value, given a cookie name. The `getCookieValue` method loops through the array of available `Cookie` objects, returning the value of any `Cookie` whose name matches the input. If there is no match, the designated default value is returned. So, for example, our typical approach for dealing with cookies is as follows:

```java
Cookie[] cookies = request.getCookies();
String color = ServletUtilities.getCookieValue(cookies, "color", "black");
String font = ServletUtilities.getCookieValue(cookies, "font", "Arial");
```

The `getCookie` method also loops through the array comparing names but returns the actual `Cookie` object instead of just the value. That method is for cases when you want to do something with the `Cookie` other than just read its value.

```java
Listing 2.17  ServletUtilities.java
```

```java
package moreservlets;
import javax.servlet.*;
import javax.servlet.http.*;

public class ServletUtilities {
    // Other parts of ServletUtilities shown elsewhere.

    /** Given an array of Cookies, a name, and a default value,
     * this method tries to find the value of the cookie with
     * the given name. If there is no cookie matching the name
     * in the array, then the default value is returned instead.
     */
```
Creating Long-Lived Cookies

Listing 2.18 shows a small class that you can use instead of Cookie if you want your cookie to automatically persist for a year when the client quits the browser. For an example of the use of this class, see the customized search engine interface of Section 8.6 of Core Servlets and JavaServer Pages (available in PDF at http://www.moreservlets.com).

```java
public static String getCookieValue(Cookie[] cookies, String cookieName, String defaultValue) {
    if (cookies != null) {
        for (int i = 0; i < cookies.length; i++) {
            Cookie cookie = cookies[i];
            if (cookieName.equals(cookie.getName()))
                return (cookie.getValue());
        }
    }
    return (defaultValue);
}

/**
 * Given an array of cookies and a name, this method tries to find and return the cookie from the array that has the given name. If there is no cookie matching the name in the array, null is returned.
 */

public static Cookie getCookie(Cookie[] cookies, String cookieName) {
    if (cookies != null) {
        for (int i = 0; i < cookies.length; i++) {
            Cookie cookie = cookies[i];
            if (cookieName.equals(cookie.getName()))
                return (cookie);
        }
    }
    return (null);
}
```
2.10 Session Tracking

This section briefly introduces the servlet session-tracking API, which keeps track of visitors as they move around at your site. For additional details and examples, see Chapter 9 of *Core Servlets and JavaServer Pages* (in PDF at http://www.moreservlets.com).

### The Need for Session Tracking

HTTP is a “stateless” protocol: each time a client retrieves a Web page, it opens a separate connection to the Web server. The server does not automatically maintain contextual information about a client. Even with servers that support persistent (keep-alive) HTTP connections and keep a socket open for multiple client requests that occur close together in time, there is no built-in support for maintaining contextual information. This lack of context causes a number of difficulties. For example, when clients at an online store add an item to their shopping carts, how does the server know what’s already in the carts? Similarly, when clients decide to proceed to checkout, how can the server determine which previously created shopping carts are theirs?

There are three typical solutions to this problem: cookies, URL rewriting, and hidden form fields. The following subsections quickly summarize what would be
required if you had to implement session tracking yourself (without using the built-in
session tracking API) each of the three ways.

Cookies
You can use HTTP cookies to store information about a shopping session, and each
subsequent connection can look up the current session and then extract information
about that session from some location on the server machine. For example, a servlet
could do something like the following:

```java
String sessionID = makeUniqueString();
Hashtable sessionInfo = new Hashtable();
Hashtable globalTable = findTableStoringSessions();
globalTable.put(sessionID, sessionInfo);
Cookie sessionCookie = new Cookie("JSESSIONID", sessionID);
sessionCookie.setPath("/");
response.addCookie(sessionCookie);
```

Then, in later requests the server could use the `globalTable` hash table to asso-
ciate a session ID from the `JSESSIONID` cookie with the `sessionInfo` hash table
of data associated with that particular session. This is an excellent solution and is the
most widely used approach for session handling. Still, it is nice that servlets have a
higher-level API that handles all this plus the following tedious tasks:

- Extracting the cookie that stores the session identifier from the other
cookies (there may be many cookies, after all).
- Setting an appropriate expiration time for the cookie.
- Associating the hash tables with each request.
- Generating the unique session identifiers.

URL Rewriting
With this approach, the client appends some extra data on the end of each URL that
identifies the session, and the server associates that identifier with data it has stored
about that session. For example, with `http://host/path/file.html;jsessionid=1234`,
the session information is attached as `jsessionid=1234`. This is also an excellent solution
and even has the advantage that it works when browsers don’t support cookies or
when the user has disabled them. However, it has most of the same problems as
cookies, namely, that the server-side program has a lot of straightforward but tedious
processing to do. In addition, you have to be very careful that every URL that refer-
ces your site and is returned to the user (even by indirect means like `Location`
fields in server redirects) has the extra information appended. And, if the user leaves
the session and comes back via a bookmark or link, the session information can be lost.
Hidden Form Fields

HTML forms can have an entry that looks like the following:

```html
<input type="hidden" name="session" value="...">
```

This entry means that, when the form is submitted, the specified name and value are included in the GET or POST data. This hidden field can be used to store information about the session but has the major disadvantage that it only works if every page is dynamically generated by a form submission. Thus, hidden form fields cannot support general session tracking; only tracking within a specific series of operations.

Session Tracking in Servlets

Servlets provide an outstanding technical solution: the HttpSession API. This high-level interface is built on top of cookies or URL rewriting. All servers are required to support session tracking with cookies, and many have a setting that lets you globally switch to URL rewriting. In fact, some servers use cookies if the browser supports them but automatically revert to URL rewriting when cookies are unsupported or explicitly disabled.

Either way, the servlet author doesn’t need to bother with many of the details, doesn’t have to explicitly manipulate cookies or information appended to the URL, and is automatically given a convenient place to store arbitrary objects that are associated with each session.

The Session-Tracking API

Using sessions in servlets is straightforward and involves looking up the session object associated with the current request, creating a new session object when necessary, looking up information associated with a session, storing information in a session, and discarding completed or abandoned sessions. Finally, if you return any URLs to the clients that reference your site and URL rewriting is being used, you need to attach the session information to the URLs.

Looking Up the HttpSession Object Associated with the Current Request

You look up the HttpSession object by calling the `getSession` method of HttpServletRequest. Behind the scenes, the system extracts a user ID from a cookie or attached URL data, then uses that as a key into a table of previously created HttpSession objects. But this is all done transparently to the programmer: you just call `getSession`. If `getSession` returns `null`, this means that the user is not already participating in a session, so you can create a new session. Creating a new session in this case is so commonly done that there is an option to automatically create a
new session if one doesn’t already exist. Just pass true to getSession. Thus, your first step usually looks like this:

```java
HttpSession session = request.getSession(true);
```

If you care whether the session existed previously or is newly created, you can use `isNew` to check.

### Looking Up Information Associated with a Session

 HttpSession objects live on the server; they’re just automatically associated with the client by a behind-the-scenes mechanism like cookies or URL rewriting. These session objects have a built-in data structure that lets you store any number of keys and associated values. In version 2.1 and earlier of the servlet API, you use `session.getValue("attribute")` to look up a previously stored value. The return type is `Object`, so you have to do a typecast to whatever more specific type of data was associated with that attribute name in the session. The return value is `null` if there is no such attribute, so you need to check for `null` before calling methods on objects associated with sessions.

In versions 2.2 and 2.3 of the servlet API, `getValue` is deprecated in favor of `getAttribute` because of the better naming match with `setAttribute` (in version 2.1, the match for `getValue` is `putValue`, not `setValue`).

Here’s a representative example, assuming `ShoppingCart` is some class you’ve defined to store information on items being purchased:

```java
HttpSession session = request.getSession(true);
ShoppingCart cart = (ShoppingCart)session.getAttribute("shoppingCart");
if (cart == null) { // No cart already in session
    cart = new ShoppingCart();
    session.setAttribute("shoppingCart", cart);
}
doSomethingWith(cart);
```

In most cases, you have a specific attribute name in mind and want to find the value (if any) already associated with that name. However, you can also discover all the attribute names in a given session by calling `getValueNames`, which returns an array of strings. This method was your only option for finding attribute names in version 2.1, but in servlet engines supporting versions 2.2 and 2.3 of the servlet specification, you can use `getAttributeNames`. That method is more consistent in that it returns an `Enumeration`, just like the `getHeaderNames` and `getParameterNames` methods of `HttpServletRequest`.
Although the data that was explicitly associated with a session is the part you care most about, some other pieces of information are sometimes useful as well. Here is a summary of the methods available in the `HttpSession` class.

```java
public Object getAttribute(String name)
public Object getValue(String name) [deprecated]
```
These methods extract a previously stored value from a session object. They return `null` if no value is associated with the given name. Use `getValue` only if you need to support servers that run version 2.1 of the servlet API. Versions 2.2 and 2.3 support both methods, but `getAttribute` is preferred and `getValue` is deprecated.

```java
public void setAttribute(String name, Object value)
public void putValue(String name, Object value) [deprecated]
```
These methods associate a value with a name. Use `putValue` only if you need to support servers that run version 2.1 of the servlet API. If the object supplied to `setAttribute` or `putValue` implements the `HttpSessionBindingListener` interface, the object's `valueBound` method is called after it is stored in the session. Similarly, if the previous value implements `HttpSessionBindingListener`, its `valueUnbound` method is called.

```java
public void removeAttribute(String name)
public void removeValue(String name) [deprecated]
```
These methods remove any values associated with the designated name. If the value being removed implements `HttpSessionBindingListener`, its `valueUnbound` method is called. Use `removeValue` only if you need to support servers that run version 2.1 of the servlet API. In versions 2.2 and 2.3, `removeAttribute` is preferred, but `removeValue` is still supported (albeit deprecated) for backward compatibility.

```java
public Enumeration getAttributeNames()
public String[] getValueNames() [deprecated]
```
These methods return the names of all attributes in the session. Use `getValueNames` only if you need to support servers that run version 2.1 of the servlet API.

```java
public String getId()
```
This method returns the unique identifier generated for each session. It is useful for debugging or logging.
public boolean isNew()
This method returns true if the client (browser) has never seen the session, usually because it was just created rather than being referenced by an incoming client request. It returns false for preexisting sessions.

public long getCreationTime()
This method returns the time in milliseconds since midnight, January 1, 1970 (GMT) at which the session was first built. To get a value useful for printing, pass the value to the Date constructor or the setTimeInMillis method of GregorianCalendar.

public long getLastAccessedTime()
This method returns the time in milliseconds since midnight, January 1, 1970 (GMT) at which the session was last sent from the client.

public int getMaxInactiveInterval()
public void setMaxInactiveInterval(int seconds)
These methods get or set the amount of time, in seconds, that a session should go without access before being automatically invalidated. A negative value indicates that the session should never time out. Note that the timeout is maintained on the server and is not the same as the cookie expiration date, which is sent to the client. See Section 5.10 (Controlling Session Timeouts) for instructions on changing the default session timeout interval.

public void invalidate()
This method invalidates the session and unbinds all objects associated with it. Use this method with caution; remember that sessions are associated with users (i.e., clients), not with individual servlets or JSP pages. So, if you invalidate a session, you might be destroying data that another servlet or JSP page is using.

Associating Information with a Session
As discussed in the previous section, you read information associated with a session by using getAttribute. To specify information, use setAttribute. To let your values perform side effects when they are stored in a session, simply have the object you are associating with the session implement the HttpSessionBindingListener interface. That way, every time setAttribute (or putValue) is called on one of those objects, its valueBound method is called immediately afterward.

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
Be aware that `setAttribute` replaces any previous values; if you want to remove a value without supplying a replacement, use `removeAttribute`. This method triggers the `valueUnbound` method of any values that implement `HttpSessionBindingListener`.

Following is an example of adding information to a session. You can add information in two ways: by adding a new session attribute (as with the first bold line in the example) or by augmenting an object that is already in the session (as in the last line of the example).

```java
HttpSession session = request.getSession(true);
ShoppingCart cart =
   (ShoppingCart)session.getAttribute("shoppingCart");
if (cart == null) { // No cart already in session
   cart = new ShoppingCart();
   session.setAttribute("shoppingCart", cart);
}
addSomethingTo(cart);
```

### Terminating Sessions

Sessions automatically become inactive when the amount of time between client accesses exceeds the interval specified by `getMaxInactiveInterval`. When this happens, any objects bound to the `HttpSession` object automatically get unbound. Then, your attached objects are automatically notified if they implement the `HttpSessionBindingListener` interface.

Rather than waiting for sessions to time out, you can explicitly deactivate a session with the session's `invalidate` method.

### Encoding URLs Sent to the Client

If you are using URL rewriting for session tracking and you send a URL that references your site to the client, you need to explicitly add the session data. There are two possible places where you might use URLs that refer to your own site.

The first is where the URLs are embedded in the Web page that the servlet generates. These URLs should be passed through the `encodeURL` method of `HttpServletResponse`. The method determines if URL rewriting is currently in use and appends the session information only if necessary. The URL is returned unchanged otherwise.

Here's an example:

```java
String originalURL = someRelativeOrAbsoluteURL;
String encodedURL = response.encodeURL(originalURL);
out.println("<A HREF=" + encodedURL + ">
</A>");
```
The second place you might use a URL that refers to your own site is in a sendRedirect call (i.e., placed into the Location response header). In this second situation, different rules determine whether session information needs to be attached, so you cannot use encodeURL. Fortunately, HttpServletResponse supplies an encodeRedirectURL method to handle that case. Here's an example:

```java
String originalURL = someURL;
String encodedURL = response.encodeRedirectURL(originalURL);
response.sendRedirect(encodedURL);
```

Since you often don't know if your servlet will later become part of a series of pages that use session tracking, it is good practice to plan ahead and encode URLs that reference your own site.

**A Servlet Showing Per-Client Access Counts**

Listing 2.19 presents a simple servlet that shows basic information about the client's session. When the client connects, the servlet uses request.getSession(true) either to retrieve the existing session or, if there was no session, to create a new one. The servlet then looks for an attribute of type Integer called accessCount. If it cannot find such an attribute, it uses 0 as the number of previous accesses. This value is then incremented and associated with the session by setAttribute. Finally, the servlet prints a small HTML table showing information about the session. Figures 2–17 and 2–18 show the servlet on the initial visit and after the page was reloaded several times.

**Listing 2.19  ShowSession.java**

```java
package moreservlets;
import java.io.*;
import javax.servlet. *;
import javax.servlet.http.*;
import java.net. *;
import java.util. *
/**
 * Simple example of session tracking. */
 public class ShowSession extends HttpServlet {
   public void doGet(HttpServletRequest request, 
                   HttpServletResponse response) 
    throws ServletException, IOException {
     response.setContentType("text/html");
     PrintWriter out = response.getWriter();
  
```
2.10 Session Tracking

Listing 2.19  ShowSession.java (continued)

String title = "Session Tracking Example";
HttpSession session = request.getSession(true);
String heading;
Integer accessCount =
    (Integer)session.getAttribute("accessCount");
if (accessCount == null) {
    accessCount = new Integer(0);
    heading = "Welcome, Newcomer";
} else {
    heading = "Welcome Back";
    accessCount = new Integer(accessCount.intValue() + 1);
}
session.setAttribute("accessCount", accessCount);
out.println(ServletUtilities.headWithTitle(title) +
"<BODY BGCOLOR="#FDF5E6">
" +
"<H1 ALIGN="CENTER">" + heading + "</H1>
" +
"<H2>Information on Your Session:</H2>
" +
"<TABLE BORDER=1 ALIGN="CENTER">
" +
"<TR BGCOLOR="#FFAD00">
" +
"<TH>Info Type</TH><TH>Value</TH>
" +
"<TR>
" +
"<TD>ID</TD><TD>" + session.getId() + "</TD>
" +
"<TR>
" +
"<TD>Creation Time</TD><TD>" +
"<TR>
" +
"<TD>Time of Last Access</TD><TD>" +
"<TR>
" +
"<TD>Number of Previous Accesses</TD><TD>" +
"</TABLE>
" +
"</BODY></HTML>";
}

/** Handle GET and POST requests identically. */

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

Source code for all examples in book: http://www.moreservlets.com/
JEE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
A Simplified Shopping Cart Application

Core Servlets and JavaServer Pages (available in PDF at http://www.moreservlets.com) presents a full-fledged shopping cart example. Most of the code in that example is for automatically building the Web pages that display the items and for the shopping cart itself. Although these application-specific pieces can be somewhat complicated, the

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
2.10 Session Tracking

basic session tracking is quite simple. This section illustrates the fundamental approach to session tracking, but without a full-featured shopping cart.

Listing 2.20 shows an application that uses a simple ArrayList (the Java 2 platform’s replacement for Vector) to keep track of all the items each user has previously purchased. In addition to finding or creating the session and inserting the newly purchased item (the value of the newItem request parameter) into it, this example outputs a bulleted list of whatever items are in the “cart” (i.e., the ArrayList). Notice that the code that outputs this list is synchronized on the ArrayList. This precaution is worth taking, but you should be aware that the circumstances that make synchronization necessary are exceedingly rare. Since each user has a separate session, the only way a race condition could occur is if the same user submits two purchases very close together in time. Although unlikely, this is possible, so synchronization is worthwhile.

Listing 2.20  ShowItems.java

```java
package moreservlets;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.ArrayList;
import moreservlets.*;

public class ShowItems extends HttpServlet {
    public void doGet(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {
        HttpSession session = request.getSession(true);
        ArrayList previousItems = (ArrayList)session.getAttribute("previousItems");
        if (previousItems == null) {
            previousItems = new ArrayList();
            session.setAttribute("previousItems", previousItems);
        }
        String newItem = request.getParameter("newItem");
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Items Purchased";
        out.println("<h1>" + title + "</h1>");
        out.println("<ul>");
```

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.coreservlets.com/.
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
Listing 2.20  ShowItems.java (continued)

```java
out.println(ServletUtilities.headWithTitle(title) +
"<BODY BGCOLOR="#FDF5E6">" +
"<H1>" + title + "</H1>";
synchronized(previousItems) {
    if (newItem != null) {
        previousItems.add(newItem);
    }
    if (previousItems.size() == 0) {
        out.println("<I>No items</I>");
    } else {
        out.println("<UL>");
        for (int i=0; i<previousItems.size(); i++) {
            out.println("<LI>" + (String)previousItems.get(i));
        }
        out.println("</UL>");
    }
}
out.println("</BODY></HTML>";)
```
2.10 Session Tracking

Source code for all examples in book: http://www.moreservlets.com/
J2EE training from the author: http://courses.corejavas.com/.
Java books, tutorials, documentation, discussion forums, and jobs: http://www.coreservlets.com/
Figure 2–21  The item display servlet after a few small purchases are made.
J2EE training from the author! Marty Hall, author of five bestselling books from Prentice Hall (including this one), is available for customized J2EE training. Distinctive features of his courses:

- Marty developed all his own course materials: no materials licensed from some unknown organization in Upper Mongolia.
- Marty personally teaches all of his courses: no inexperienced flunky regurgitating memorized PowerPoint slides.
- Marty has taught thousands of developers in the USA, Canada, Australia, Japan, Puerto Rico, and the Philippines: no first-time instructor using your developers as guinea pigs.
- Courses are available onsite at your organization (US and internationally): cheaper, more convenient, and more flexible. Customizable content!
- Courses are also available at public venues: for organizations without enough developers for onsite courses.
- Many topics are available: intermediate servlets & JSP, advanced servlets & JSP, Struts, JSF, Java 5, AJAX, and more. Custom combinations of topics are available for onsite courses.

Need more details? Want to look at sample course materials? Check out http://courses.coreservlets.com/. Want to talk directly to the instructor about a possible course? Email Marty at hall@coreservlets.com.