Servlet Basics

Topics in This Chapter

- The basic structure of servlets
- A simple servlet that generates plain text
- A servlet that generates HTML
- Servlets and packages
- Some utilities that help build HTML
- The servlet life cycle
- How to deal with multithreading problems
- Tools for interactively talking to servlets
- Servlet debugging strategies

Chapter 3

As discussed in Chapter 1, servlets are programs that run on a Web or application server and act 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 3–1.

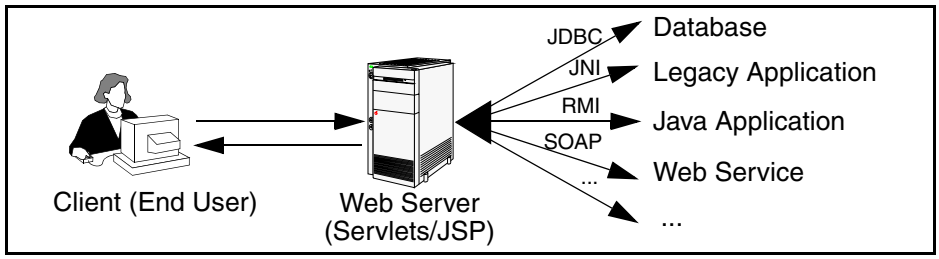


Figure 3–1 The role of Web middleware.

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 3–1 shows a single arrow going from the client to the Web server (the layer in which servlets and JSP pages 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 types of data are critical to effective development. The HTTP information includes cookies, media types and compression schemes the browser understands, and so forth; it is discussed in Chapter 5.

3. **Generate the results.**
   This process may require talking to a database, executing an RMI or CORBA call, invoking a Web service, 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 or XML), binary (GIF images), Excel, 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 3–1 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 types of data 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. These tasks are discussed in Chapters 6–8.

In principle, servlets are not restricted to Web or application servers that handle HTTP requests but can be used for other types of servers as well. For example, servlets could be embedded in FTP or mail servers to extend their functionality. In practice, however, this use of servlets has not caught on, and we discuss only HTTP servlets.

3.1 Basic Servlet Structure

Listing 3.1 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 the use of HTML forms and the distinctions between GET and POST, see Chapter 19 (Creating and Processing HTML Forms).

Listing 3.1 ServletTemplate.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.
  }
}
```

Servlets typically 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 lets you get at all of the *incoming* data; the class has methods by which you can find out about 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, HttpServletResponse lets you obtain a PrintWriter that you use to send 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 chapters.

Since `doGet` and `doPost` throw two exceptions (`ServletException` and `IOException`), you are required to include them in the method 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`).

However, there is no need to memorize the method signature and import statements. Instead, simply download the preceding template from the source code archive at http://www.coreservlets.com/ and use it as a starting point for your servlets.

3.2 A Servlet That Generates Plain Text

Listing 3.2 shows a simple servlet that outputs plain text, with the output shown in Figure 3–2. Before we move on, it is worth spending some time reviewing the process of installing, compiling, and running this simple servlet. See Chapter 2 (Server Setup and Configuration) for a much more detailed description of the process.

First, be sure that you've already verified the basics:

- That your server is set up properly as described in Section 2.3 (Configure the Server).
- That your development `CLASSPATH` refers to the necessary three entries (the servlet JAR file, your top-level development directory, and ".") as described in Section 2.7 (Set Up Your Development Environment).
- That all of the test cases of Section 2.8 (Test Your Setup) execute successfully.

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 step will compile your servlet to create HelloWorld.class.

Third, move HelloWorld.class to the directory that your server uses to store servlets that are in the default Web application. The exact location varies from server to server, but is typically of the form *install_dir*/.../WEB-INF/classes (see Section 2.10 for details). For Tomcat you use *install_dir*/webapps/ROOT/WEB-INF/classes, for JRun you use *install_dir*/servers/default/default-ear/default-war/WEB-INF/classes, and for Resin you use *install_dir*/doc/WEB-INF/classes. Alternatively, you can use one of the techniques of Section 2.9 (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 2.11 (Web Applications: A Preview). During initial development, you will almost certainly find it convenient to use the default URL so that you don't have to edit the web.xml file each time you test a new servlet. When you deploy real applications, however, you almost always disable the default URL and assign explicit URLs in the web.xml file (see Section 2.11, "Web Applications: A Preview"). In fact, servers are not absolutely required to support the default URL, and a few, most notably BEA WebLogic, do not.

Figure 3–2 shows the servlet being accessed by means of the default URL, with the server running on the local machine.

Listing 3.2 HelloWorld.java

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

public class HelloWorld extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    PrintWriter out = response.getWriter();
    out.println("Hello World");
  }
}
```

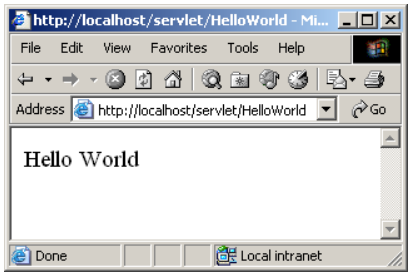


Figure 3–2 Result of http://localhost/servlet/HelloWorld.

3.3 A Servlet That Generates HTML

Most servlets generate HTML, not plain text as in the previous example. To generate HTML, you add three steps to the process just shown:

1. Tell the browser that you're sending it HTML.
2. Modify the `println` statements to build a legal Web page.
3. Check your HTML with a formal syntax validator.

You accomplish the first step by setting the HTTP `Content-Type` response header to `text/html`. 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:

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

Although HTML is the most common kind 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 Excel spreadsheets (content type `application/vnd.ms-excel`—see Section 7.3), JPEG images (content type `image/jpeg`—see Section 7.5), and XML documents (content type `text/xml`). Also, you rarely use servlets to generate HTML pages that have relatively fixed formats (i.e., whose layout changes little for each request); JSP is usually more convenient in such a case. JSP is discussed in Part II of this book (starting in Chapter 10).

Don't be concerned if you are not yet familiar with HTTP response headers; they are discussed in Chapter 7. However, you should note now 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 best approach is to simply put the `setContentType` line before any of the lines that use the `PrintWriter`.

Core Warning

*You must 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 3.3 shows HelloServlet.java, the sample servlet used in Section 2.8 to verify that the server is functioning properly. As Figure 3–3 illustrates, the browser formats the result as HTML, not as plain text.

Listing 3.3 HelloServlet.java

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

/** Simple servlet used to test server. */

public class HelloServlet 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>\n" +
                "<HEAD><TITLE>Hello</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1>Hello</H1>\n" +
                "</BODY></HTML>");
  }
}
```

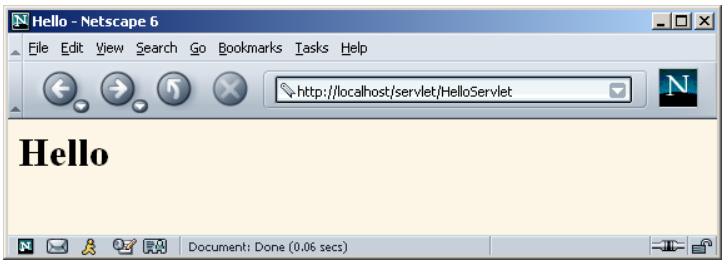


Figure 3–3 Result of http://localhost/servlet/HelloServlet.

The final step is to check that your HTML has no syntax errors that could cause unpredictable results on different browsers. See Section 3.5 (Simple HTML-Building Utilities) for a discussion of HTML validators.

3.4 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 inadvertently choose the same name for a servlet or a utility class. Now, Web applications (see Section 2.11) help with this problem by dividing things up into separate directories, each with its own set of servlets, utility classes, JSP pages, and HTML files. However, since even a single Web application can be large, you still need the standard Java solution for avoiding name conflicts: packages. Besides, as you will see later, custom classes used by JSP pages should *always* be in packages. You might as well get in the habit early.

When you put your servlets in packages, you need to perform the following two additional steps.

1. **Place the files in a subdirectory that matches the intended package name.** For example, we'll use the `coreservlets` package for most of the rest of the servlets in this book. So, the class files need to go in a subdirectory called coreservlets. Remember that case matters for both package names and directory names, regardless of what operating system you are using.
2. **Insert a package statement in the class file.** For instance, for a class to be in a package called `somePackage`, the class should be in the somePackage directory and the *first* non-comment line of the file should read

```
package somePackage;
```

For example, Listing 3.4 presents a variation of the `HelloServlet` class that is in the `coreservlets` package and thus the coreservlets directory. As discussed in Section 2.8 (Test Your Setup), the class file should be placed in *install_dir*/webapps/ROOT/WEB-INF/classes/coreservlets for Tomcat, *install_dir*/servers/default/default-ear/default-war/WEB-INF/classes/coreservlets for JRun, and *install_dir*/doc/WEB-INF/classes/coreservlets for Resin. Other servers have similar installation locations.

Figure 3–4 shows the servlet accessed by means of the default URL.

Listing 3.4 coreservlets/HelloServlet2.java

```
package coreservlets;

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

/** Simple servlet for testing 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>\n" +
                "<HEAD><TITLE>Hello (2)</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1>Hello (2)</H1>\n" +
                "</BODY></HTML>");
  }
}
```

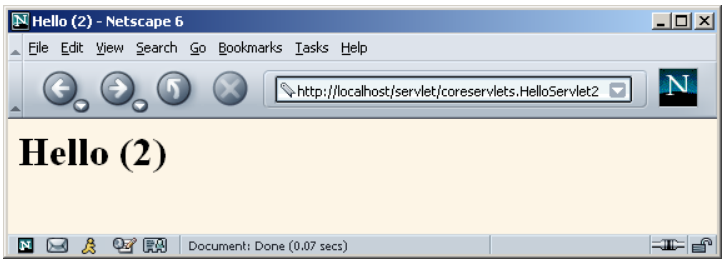


Figure 3–4 Result of http://localhost/servlet/coreservlets.HelloServlet2.

3.5 Simple HTML-Building Utilities

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

```
<!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 specifications require it. We 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 client, 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.

Core Approach

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

Admittedly, it is sometimes a bit cumbersome to generate HTML with `println` statements, especially long tedious lines like the `DOCTYPE` declaration. Some people address this problem by writing lengthy HTML-generation utilities, then use the utilities throughout their servlets. We're 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 (see Chapter 10, "Overview of JSP Technology"). 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 class that simplifies those constructs. After all, you're working with the Java programming language; don't forget the standard object-oriented programming principle of reusing, not repeating, code. Repeating identical or nearly identical code means that you have to change the code lots of different places when you inevitably change your approach.

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 3.5, with Listing 3.6 showing a variation of the `HelloServlet` class that makes use of this utility. We'll add a few more utilities throughout the book.

Listing 3.5 coreservlets/ServletUtilities.java

```
package coreservlets;

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\">";
```

Listing 3.5 coreservlets/ServletUtilities.java *(continued)*

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

   ...
}
```

Listing 3.6 coreservlets/HelloServlet3.java

```
package coreservlets;

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

/** Simple servlet for testing the use of packages
 *  and utilities from the same package.
 */

public class HelloServlet3 extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Hello (3)";
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1>" + title + "</H1>\n" +
                "</BODY></HTML>");
  }
}
```

After you compile HelloServlet3.java (which results in ServletUtilities.java being compiled automatically), you need to move the two class files to the coreservlets subdirectory of the server's default deployment location (.../WEB-INF/classes; review Section 2.8 for details). If you get an "Unresolved symbol" error when compiling HelloServlet3.java, go back and review the CLASSPATH settings described in Section 2.7 (Set Up Your Development Environment), especially the part about including the top-level development directory in the CLASSPATH. Figure 3–5 shows the result when the servlet is invoked with the default URL.

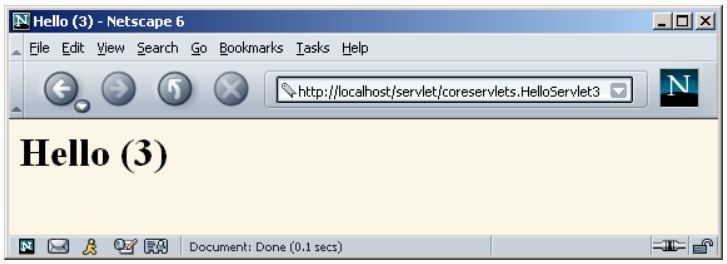


Figure 3–5 Result of http://localhost/servlet/coreservlets.HelloServlet3.

3.6 The Servlet Life Cycle

In Section 1.4 (The Advantages of Servlets Over "Traditional" CGI) we referred to the fact that only a single instance of each servlet gets created, with each user request resulting in a new thread that is handed off to `doGet` or `doPost` as appropriate. We'll now be more specific about how servlets are created and destroyed, and how and when the various methods are invoked. We summarize 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, if the server decides to unload a servlet, it first calls the servlet's `destroy` method.

The service Method

Each time the server receives a request for a servlet, the server spawns a new thread 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 19 (Creating and Processing HTML Forms).

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.

```
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, as illustrated in Listing 3.7. Since getLastModified is invoked by the default service method, overriding service eliminates this option. Finally, service gives you 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.

Normally, you do not need to implement doHead in order to handle HEAD requests (HEAD requests stipulate that the server should return the normal HTTP headers, but no associated document). You don't normally need to implement doHead because the system automatically calls doGet and uses the resultant status line and header settings to answer HEAD requests. However, it is occasionally useful

to implement doHead 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.

The init Method

Most of the time, your servlets deal only with per-request data, and doGet or doPost are the only life-cycle methods you need. Occasionally, however, you want to perform complex setup tasks when the servlet is first loaded, but not repeat those tasks for each request. The init method is designed for this case; it is called when the servlet is first created, and *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 the chapter on the web.xml file in Volume 2 of this book).

The init method definition looks like this:

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

The init method performs two varieties of initializations: general initializations and initializations controlled by initialization parameters.

General Initializations

With the first type of initialization, init simply creates or loads some data that will be used throughout the life of the servlet, or it performs some one-time computation. If you are familiar with applets, this task is analogous to an applet calling getImage to load image files over the network: the operation only needs to be performed once, so it is triggered by init. Servlet examples include setting up a database connection pool for requests that the servlet will handle or loading a data file into a HashMap.

Listing 3.7 shows a servlet that uses init to do two things.

First, it builds an array of 10 integers. Since these numbers are based upon complex calculations, we don't want to repeat the computation for each request. So, doGet looks up the values that init computed, instead of generating them each time. The results of this technique are shown in Figure 3–6.

Second, since the output of the servlet does not change except when the server is rebooted, init also stores a page modification date that is used by the getLastModified method. This method should return a modification time expressed in milliseconds since 1970, as is standard with Java dates. The time is automatically converted to a date in GMT appropriate for the Last-Modified header. More importantly, if the server receives a conditional GET request (one specifying that the client only wants pages marked If-Modified-Since a particular date), the system

compares the specified date to that returned by `getLastModified`, returning the page only if it has been changed after the specified date. Browsers frequently make these conditional requests for pages stored in their caches, so supporting conditional requests helps your users (they get faster results) and reduces server load (you send fewer complete documents). Since the `Last-Modified` and `If-Modified-Since` headers use only whole seconds, the `getLastModified` method should round times down to the nearest second.

Listing 3.7 coreservlets/LotteryNumbers.java

```
package coreservlets;

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

/** Example using servlet initialization and the
 *  getLastModified method.
 */

public class LotteryNumbers extends HttpServlet {
  private long modTime;
  private int[] numbers = new int[10];

  /** The init method is called only when the servlet is first
   *  loaded, before the first request is processed.
   */

  public void init() throws ServletException {
    // Round to nearest second (i.e., 1000 milliseconds)
    modTime = System.currentTimeMillis()/1000*1000;
    for(int i=0; i<numbers.length; i++) {
      numbers[i] = randomNum();
    }
  }

 /** Return the list of numbers that init computed. */

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Your Lottery Numbers";
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
```

Listing 3.7 coreservlets/LotteryNumbers.java *(continued)*

```
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                "<B>Based upon extensive research of " +
                "astro-illogical trends, psychic farces, " +
                "and detailed statistical claptrap, " +
                "we have chosen the " + numbers.length +
                " best lottery numbers for you.</B>" +
                "<OL>");
    for(int i=0; i<numbers.length; i++) {
      out.println("  <LI>" + numbers[i]);
    }
    out.println("</OL>" +
                "</BODY></HTML>");
  }

  /** The standard service method compares this date against
   *  any date specified in the If-Modified-Since request header.
   *  If the getLastModified date is later or if there is no
   *  If-Modified-Since header, the doGet method is called
   *  normally. But if the getLastModified date is the same or
   *  earlier, the service method sends back a 304 (Not Modified)
   *  response and does <B>not</B> call doGet. The browser should
   *  use its cached version of the page in such a case.
   */

  public long getLastModified(HttpServletRequest request) {
    return(modTime);
  }

  // A random int from 0 to 99.

  private int randomNum() {
    return((int)(Math.random() * 100));
  }
}
```

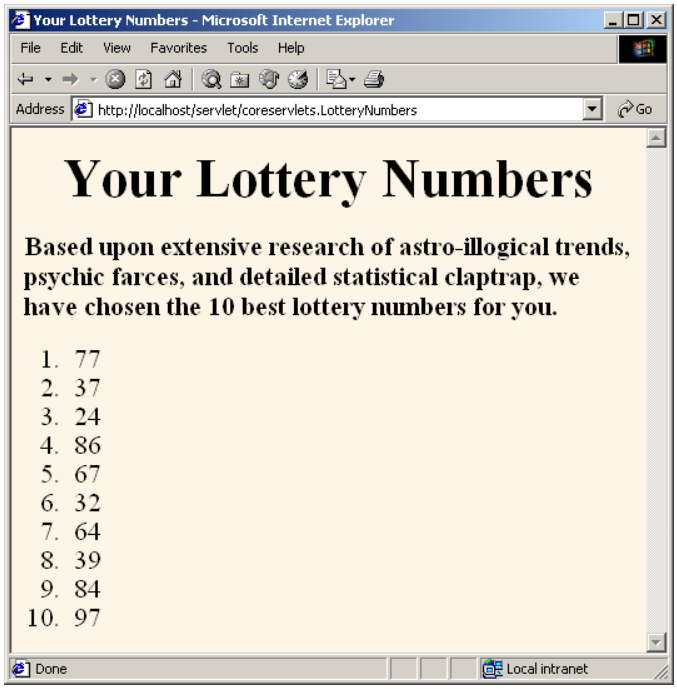


Figure 3–6 Result of the `LotteryNumbers` servlet.

Figures 3–7 and 3–8 show the result of requests for the same servlet with two slightly different `If-Modified-Since` dates. To set the request headers and see the response headers, we used `WebClient`, a Java application that lets you interactively set up HTTP requests, submit them, and see the "raw" results. The code for `WebClient` is available at the source code archive on the book's home page (http://www.coreservlets.com/).

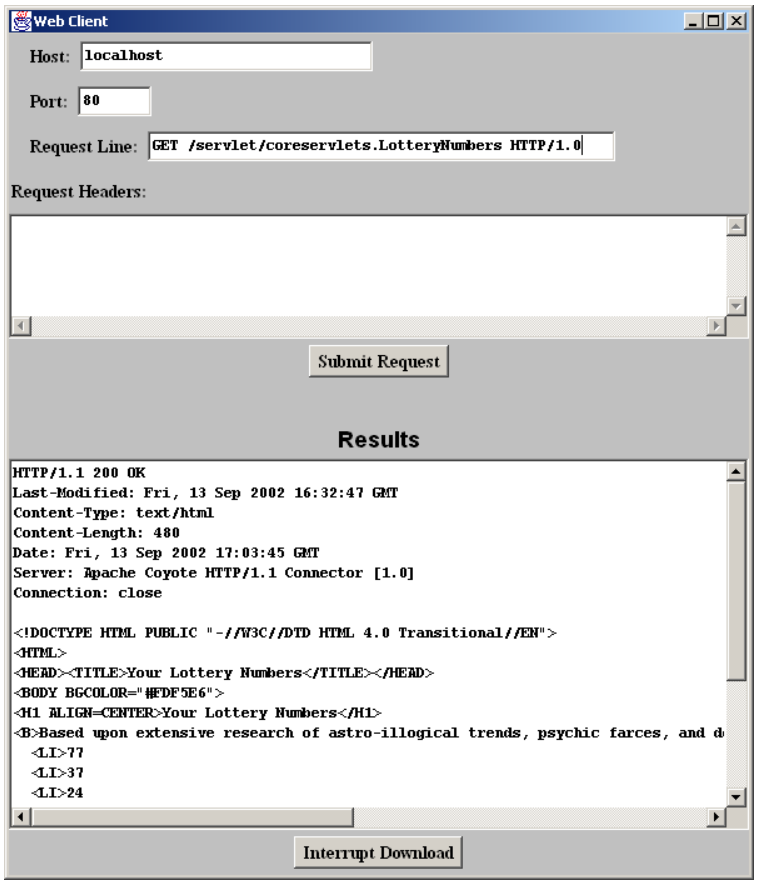


Figure 3–7 Accessing the `LotteryNumbers` servlet results in normal response (with the document sent to the client) in two situations: when there is an unconditional `GET` request or when there is a conditional request that specifies a date before servlet initialization. Code for the `WebClient` program (used here to interactively connect to the server) is available at the book's source code archive at http://www.coreservlets.com/.

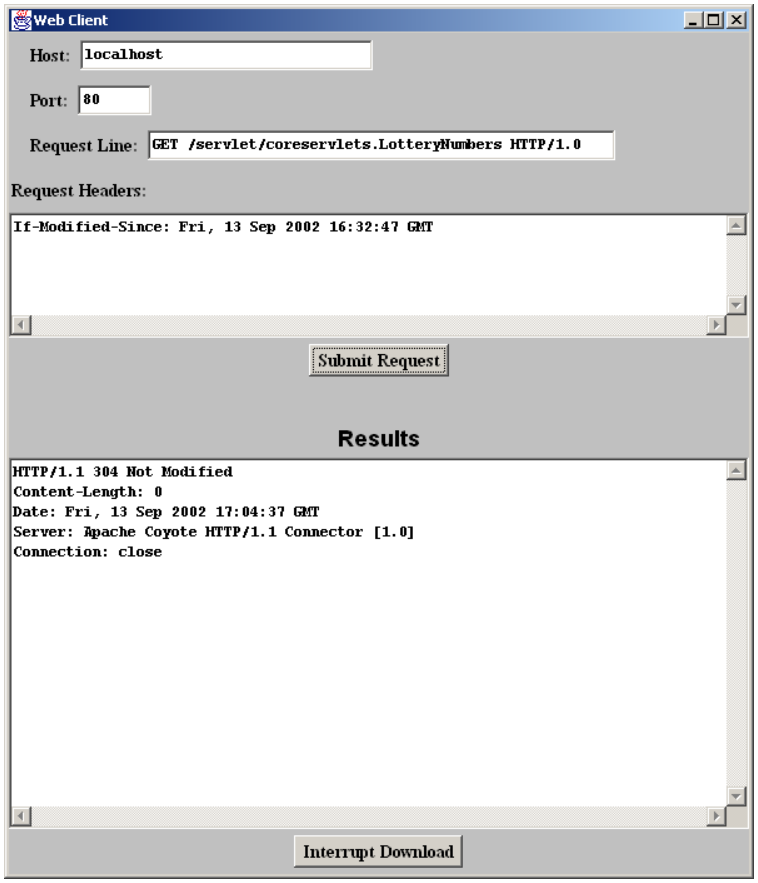


Figure 3–8 Accessing the `LotteryNumbers` servlet results in a 304 (Not Modified) response with no actual document in one situation: when a conditional `GET` request is received that specifies a date at or after servlet initialization.

Initializations Controlled by Initialization Parameters

In the previous example, the `init` method computed some data that was used by the `doGet` and `getLastModified` methods. Although this type of general initialization is quite common, it is also common to control the initialization by the use of initialization parameters. To understand the motivation for init parameters, you need to understand the categories of people who might want to customize the way a servlet or JSP page behaves. There are three such groups:

1. Developers.
2. End users.
3. Deployers.

Developers change the behavior of a servlet by changing the code. End users change the behavior of a servlet by providing data to an HTML form (assuming that the developer has written the servlet to look for this data). But what about deployers? There needs to be a way to let administrators move servlets from machine to machine and change certain parameters (e.g., the address of a database, the size of a connection pool, or the location of a data file) without modifying the servlet source code. Providing this capability is the purpose of init parameters.

Because the use of servlet initialization parameters relies heavily on the deployment descriptor (web.xml), we postpone details and examples on init parameters until the deployment descriptor chapter in Volume 2 of this book. But, here is a brief preview:

1. Use the web.xml `servlet` element to give a name to your servlet.
2. Use the web.xml `servlet-mapping` element to assign a custom URL to your servlet. You never use default URLs of the form http://.../**servlet**/*ServletName* when using init parameters. In fact, these default URLs, although extremely convenient during initial development, are almost never used in deployment scenarios.
3. Add `init-param` subelements to the web.xml `servlet` element to assign names and values of initialization parameters.
4. From within your servlet's `init` method, call `getServletConfig` to obtain a reference to the `ServletConfig` object.
5. Call the `getInitParameter` method of `ServletConfig` with the name of the init parameter. The return value is the value of the init parameter or `null` if no such init parameter is found in the web.xml file.

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 (remember those California power outages?). So, don't count on `destroy` as the *only* mechanism for saving state to disk. If your servlet performs activities like counting hits or accumulating lists of cookie values that indicate special access, you should also proactively write the data to disk periodically.

3.7 The SingleThreadModel Interface

Normally, the system makes a single instance of your servlet and then creates a new thread for each user request. This means that if a new request comes in while a previous request is still executing, multiple threads can concurrently be accessing the same servlet object. Consequently, your `doGet` and `doPost` methods must be careful to synchronize access to fields and other shared data (if any) since multiple threads may access the data simultaneously. Note that local variables are not shared by multiple threads, and thus need no special protection.

In principle, you can prevent multithreaded access by having your servlet implement the `SingleThreadModel` interface, as below.

```
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.

Although `SingleThreadModel` prevents concurrent access in principle, in practice there are two reasons why it is usually a poor choice.

First, 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 cannot handle pending requests for the same servlet. So, think twice before using the `SingleThreadModel` approach. Instead, consider synchronizing only the part of the code that manipulates the shared data.

The second problem with `SingleThreadModel` stems from the fact that the specification permits servers to use pools of instances instead of queueing up the requests to a single instance. As long as each instance handles only one request at a time, the pool-of-instances approach satisfies the requirements of the specification. But, it is a bad idea.

Suppose, on one hand, that you are using regular non-static instance variables (fields) to refer to shared data. Sure, `SingleThreadModel` prevents concurrent access, but it does so by throwing out the baby with the bath water: each servlet instance has a separate copy of the instance variables, so the data is no longer shared properly.

On the other hand, suppose that you are using static instance variables to refer to the shared data. In that case, the pool-of-instances approach to `SingleThreadModel` provides no advantage whatsoever; multiple requests (using different instances) can still concurrently access the static data.

Now, `SingleThreadModel` is still occasionally useful. For example, it can be used when the instance variables are reinitialized for each request (e.g., when they are used merely to simplify communication among methods). But, the problems with `SingleThreadModel` are so severe that it is deprecated in the servlet 2.4 (JSP 2.0) specification. You are much better off using explicit `synchronized` blocks.

Core Warning

Avoid implementing `SingleThreadModel` for high-traffic servlets. Use it with great caution at other times. For production-level code, explicit code synchronization is almost always better. `SingleThreadModel` is deprecated in version 2.4 of the servlet specification.

For example, consider the servlet of Listing 3.8 that attempts to assign unique user IDs to each client (unique until the server restarts, that is). It uses an instance variable (field) called `nextID` to keep track of which ID should be assigned next, and uses the following code to output the ID.

```
String id = "User-ID-" + nextID;
out.println("<H2>" + id + "</H2>");
nextID = nextID + 1;
```

Now, suppose you were very careful in testing this servlet. You put it in a subdirectory called coreservlets, compiled it, and copied the coreservlets directory to the WEB-INF/classes directory of the default Web application (see Section 2.10, "Deployment Directories for Default Web Application: Summary"). You started the server. You repeatedly accessed the servlet with http://localhost/servlet/coreservlets.UserIDs. Every time you accessed it, you got a different value (Figure 3–9). So the code is correct, right? Wrong! The problem occurs only when there are multiple simultaneous accesses to the servlet. Even then, it occurs only once in a while. But, in a few cases, the first client could read the `nextID` field and have its thread preempted before it incremented the field. Then, a second client could read the field and get the same value as the first client. Big trouble! For example, there have been real-world e-commerce applications where customer purchases were occasionally charged to the wrong client's credit card, precisely because of such a race condition in the generation of user IDs.

Now, if you are familiar with multithreaded programming, the problem was very obvious to you. The question is, what is the proper solution? Here are three possibilities.

1. **Shorten the race.** Remove the third line of the code snippet and change the first line to the following.

   ```
   String id = "User-ID-" + nextID++;
   ```

   Boo! This approach decreases the likelihood of an incorrect answer, but does not eliminate the possibility. In many scenarios, lowering the probability of a wrong answer is a bad thing, not a good thing: it merely means that the problem is less likely to be detected in testing, and more likely to occur after being fielded.
2. **Use `SingleThreadModel`.** Change the servlet class definition to the following.

   ```
   public class UserIDs extends HttpServlet
                        implements SingleThreadModel {
   ```

   Will this work? If the server implements `SingleThreadModel` by queueing up all the requests, then, yes, this will work. But at a performance cost if there is a lot of concurrent access. Even worse, if the server implements `SingleThreadModel` by making a pool of servlet instances, this approach will totally fail because each instance will have its own `nextID` field. Either server implementation approach is legal, so this "solution" is no solution at all.
3. **Synchronize the code explicitly.** Use the standard synchronization construct of the Java programming language. Start a `synchronized` block just before the first access to the shared data, and end the block just after the last update to the data, as follows.

   ```
   synchronized(this) {
     String id = "User-ID-" + nextID;
     out.println("<H2>" + id + "</H2>");
     nextID = nextID + 1;
   }
   ```

   This technique tells the system that, once a thread has entered the above block of code (or any other `synchronized` section labelled with the same object reference), no other thread is allowed in until the first thread exits. This is the solution you have always used in the Java programming language. It is the right one here, too. Forget error-prone and low-performance `SingleThreadModel` shortcuts; fix race conditions the right way.

Listing 3.8 coreservlets/UserIDs.java

```
package coreservlets;

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

/** Servlet that attempts to give each user a unique
 *  user ID. However, because it fails to synchronize
 *  access to the nextID field, it suffers from race
 *  conditions: two users could get the same ID.
 */

public class UserIDs extends HttpServlet {
  private int nextID = 0;

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Your ID";
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                "<CENTER>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1>" + title + "</H1>\n");
    String id = "User-ID-" + nextID;
    out.println("<H2>" + id + "</H2>");
    nextID = nextID + 1;
    out.println("</BODY></HTML>");
  }
}
```

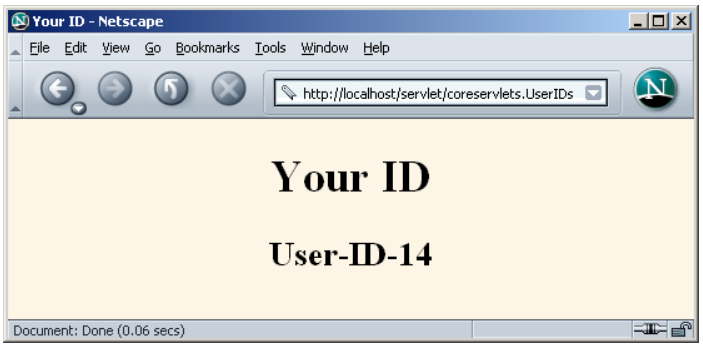


Figure 3–9 Result of the `UserIDs` servlet.

3.8 Servlet Debugging

Naturally, when *you* write servlets, you never make mistakes. However, some of your colleagues might make an occasional error, and you can pass this advice on to them. Seriously, though, debugging servlets can be tricky because you don't execute them directly. Instead, you trigger their execution by means of an HTTP request, and they are executed by the Web server. This remote execution makes it difficult to insert break points or to read debugging messages and stack traces. So, approaches to servlet debugging differ somewhat from those used in general development. Here are 10 general strategies that can make your life easier.

1. **Use print statements.**
   With most server vendors, if you run the server on your desktop, a window pops up that displays standard output (i.e., the result of `System.out.println` statements). "What?" you say, "Surely you aren't advocating something as old-fashioned as print statements?" Well, true, there are more sophisticated debugging techniques. And if you are familiar with them, by all means use them. But you'd be surprised how useful it is to just gather basic information about how your program is operating. The `init` method doesn't seem to work? Insert a print statement, restart the server, and see if the print statement is displayed in the standard output window. Perhaps you declared `init` incorrectly, so your version isn't being called? Get a `NullPointerException`? Insert a couple of print statements to find out which line of code generated the error and which object on that line was `null`. When in doubt, gather more information.

2. **Use an integrated debugger in your IDE.**
   Many integrated development environments (IDEs) have sophisticated debugging tools that can be integrated with your servlet and JSP container. The Enterprise editions of IDEs like Borland JBuilder, Oracle JDeveloper, IBM WebSphere Studio, Eclipse, BEA WebLogic Studio, Sun ONE Studio, etc., typically let you insert breakpoints, trace method calls, and so on. Some will even let you connect to a server running on a remote system.
3. **Use the log file.**
   The `HttpServlet` class has a method called `log` that lets you write information into a logging file on the server. Reading debugging messages from the log file is a bit less convenient than watching them directly from a window as with the two previous approaches, but using the log file is an option even when running on a remote server; in such a situation, print statements are rarely useful and only the advanced IDEs support remote debugging. The `log` method has two variations: one that takes a `String`, and the other that takes a `String` and a `Throwable` (an ancestor class of `Exception`). The exact location of the log file is server-specific, but is generally clearly documented or can be found in subdirectories of the server installation directory.
4. **Use Apache Log4J.**
   Log4J is a package from the Apache Jakarta Project—the same project that manages Tomcat (one of the sample servers used in the book) and Struts (an MVC framework discussed in Volume 2 of this book). With Log4J, you semi-permanently insert debugging statements in your code and use an XML-based configuration file to control which are invoked at request time. Log4J is fast, flexible, convenient, and becoming more popular by the day. For details, see http://jakarta.apache.org/log4j/.
5. **Write separate classes.**
   One of the basic principles of good software design is to put commonly used code into a separate function or class so you don't need to keep rewriting it. That principle is even more important when you are writing servlets, since these separate classes can often be tested independently of the server. You can even write a test routine, with a `main`, that can be used to generate hundreds or thousands of test cases for your routines—not something you are likely to do if you have to submit each test case by hand in a browser.
6. **Plan ahead for missing or malformed data.**
   Are you reading form data from the client (Chapter 4)? Remember to check whether it is `null` or an empty string. Are you processing HTTP request headers (Chapter 5)? Remember that the headers are optional and thus might be `null` in any particular request. Every time

you
process data that comes directly or indirectly from a client, be sure to consider the possibility that it was entered incorrectly or omitted altogether.

7. **Look at the HTML source.**
   If the result you see in the browser looks odd, choose View Source from the browser's menu. Sometimes a small HTML error like `<TABLE>` instead of `</TABLE>` can prevent much of the page from being viewed. Even better, use a formal HTML validator on the servlet's output. See Section 3.5 (Simple HTML-Building Utilities) for a discussion of this approach.
8. **Look at the request data separately.**
   Servlets read data from the HTTP request, construct a response, and send it back to the client. If something in the process goes wrong, you want to discover if the cause is that the client is sending the wrong data or that the servlet is processing it incorrectly. The `EchoServer` class, discussed in Chapter 19 (Creating and Processing HTML Forms), lets you submit HTML forms and get a result that shows you *exactly* how the data arrived at the server. This class is merely a simple HTTP server that, for all requests, constructs an HTML page showing what was sent. Full source code is online at http://www.coreservlets.com/.
9. **Look at the response data separately.**
   Once you look at the request data separately, you'll want to do the same for the response data. The `WebClient` class, discussed in the `init` example of Section 3.6 (The Servlet Life Cycle), lets you connect to the server interactively, send custom HTTP request data, and see everything that comes back—HTTP response headers and all. Again, you can download the source code from http://www.coreservlets.com/.
10. **Stop and restart the server.**
    Servers are supposed to keep servlets in memory between requests, not reload them each time they are executed. However, most servers support a development mode in which servlets are supposed to be automatically reloaded whenever their associated class file changes. At times, however, some servers can get confused, especially when your only change is to a lower-level class, not to the top-level servlet class. So, if it appears that changes you make to your servlets are not reflected in the servlet's behavior, try restarting the server. Similarly, the `init` method is run only when a servlet is first loaded, the web.xml file (see Section 2.11) is read only when a Web application is first loaded (although many servers have a custom extension for reloading it), and certain Web application listeners (see Volume 2) are triggered only when the server first starts. Restarting the server will simplify debugging in all of those situations.

Handling the Client Request: Form Data

Topics in This Chapter

- Reading individual request parameters
- Reading the entire set of request parameters
- Handling missing and malformed data
- Filtering special characters out of the request parameters
- Automatically filling in a data object with request parameter values
- Dealing with incomplete form submissions

Chapter 4

One of the main motivations for building Web pages dynamically is so that the result can be based upon user input. This chapter shows you how to access that input (Sections 4.1–4.4). It also shows you how to use default values when some of the expected parameters are missing (Section 4.5), how to filter < and > out of the request data to avoid messing up the HTML results (Section 4.6), how to create "form beans" that can be automatically populated from the request data (Section 4.7), and how, when required request parameters are missing, to redisplay the form with the missing values highlighted (Section 4.8).

4.1 The Role of Form Data

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=sfo. The part after the question mark (i.e., user=Marty+Hall&origin=bwi&dest=sfo) 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; form data can also be sent to the server on a separate line for POST requests. If you're not familiar with

HTML forms, Chapter 19 (Creating and Processing HTML Forms) gives details on how to build forms that collect and transmit data of this sort. However, here are the basics.

1. **Use the `FORM` element to create an HTML form.** Use the `ACTION` attribute to designate the address of the servlet or JSP page that will process the results; you can use an absolute or relative URL. For example:

   ```
   <FORM ACTION="...">...</FORM>
   ```

   If `ACTION` is omitted, the data is submitted to the URL of the current page.
2. **Use input elements to collect user data.** Place the elements between the start and end tags of the `FORM` element and give each input element a `NAME`. Textfields are the most common input element; they are created with the following.

   ```
   <INPUT TYPE="TEXT" NAME="...">
   ```

3. **Place a submit button near the bottom of the form.** For example:

   ```
   <INPUT TYPE="SUBMIT">
   ```

   When the button is pressed, the URL designated by the form's `ACTION` is invoked. With `GET` requests, a question mark and name/value pairs are attached to the end of the URL, where the names come from the `NAME` attributes in the HTML input elements and the values come from the end user. With `POST` requests, the same data is sent, but on a separate request line instead of attached to the URL.

Extracting the needed information from this form data is traditionally one of the most tedious parts of server-side programming.

First of all, before servlets you generally had 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: reverse the encoding that the browser uses on certain characters. Alphanumeric characters are sent unchanged by the browser, 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. For example, if someone enters a value of "`~hall, ~gates, and ~mcnealy`" into a textfield with the name `users` in an HTML form, the data is sent as "`users=%7Ehall%2C+%7Egates%2C+and+%7Emcnealy`", and the server-side program has to reconstitute the original string.

Finally, the fourth reason that it is tedious to parse form data with traditional server-side technologies is that values can be omitted (e.g., "`param1=val1&`**`param2=`**`&param3=val3`") or a parameter can appear more than once (e.g., "**`param1=val1`**`&param2=val2&`**`param1=val3`**"), so your parsing code needs special cases for these situations.

Fortunately, servlets help us with much of this tedious parsing. That's the topic of the next section.

4.2 Reading Form Data from Servlets

One of the nice features of servlets is that all of this form parsing is handled automatically. You call `request.getParameter` to get the value of a form parameter. You can also call `request.getParameterValues` if the parameter appears more than once, or you can call `request.getParameterNames` if you want a complete list of all parameters in the current request. In the rare cases in which you need to read the raw request data and parse it yourself, call `getReader` or `getInputStream`.

Reading Single Values: getParameter

To read a request (form) parameter, you simply call the `getParameter` method of `HttpServletRequest`, supplying the case-sensitive parameter name as an argument. You supply the parameter name exactly as it appeared in the HTML source code, and you get the result exactly as the end user entered it; any necessary URL-decoding is done automatically. Unlike the case with many alternatives to servlet technology, you use `getParameter` exactly the same way when the data is sent by `GET` (i.e., from within the `doGet` method) as you do when it is sent by `POST` (i.e., from within `doPost`); the servlet knows which request method the client used and automatically uses the appropriate method to read the data. An empty `String` is returned if the parameter exists but has no value (i.e., the user left the corresponding textfield empty when submitting the form), and `null` is returned if there was no such parameter.

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

Core Warning

The values supplied to `getParameter` and `getParameterValues` are case sensitive.

Reading Multiple Values: getParameterValues

If the same parameter name might appear in the form data more than once, you should call `getParameterValues` (which returns an array of strings) instead of `getParameter` (which returns a single string corresponding to the first occurrence of the parameter). 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.

Now, if you are the author of the HTML form, it is usually best to ensure that each textfield, checkbox, or other user interface element has a unique name. That way, you can just stick with the simpler `getParameter` method and avoid `getParameterValues` altogether. However, you sometimes write servlets or JSP pages that handle other people's HTML forms, so you have to be able to deal with all possible cases. Besides, multiselectable list boxes (i.e., HTML `SELECT` elements with the `MULTIPLE` attribute set; see Chapter 19 for details) repeat the parameter name for each selected element in the list. So, you cannot always avoid multiple values.

Looking Up Parameter Names: getParameterNames and getParameterMap

Most servlets look for a specific set of parameter names; in most cases, if the servlet does not know the name of the parameter, it does not know what to do with it either. So, your primary tool should be `getParameter`. However, it is sometimes useful to get a full list of parameter names. The primary utility of the full list is debugging, but you occasionally use the list for applications where the parameter names are very dynamic. For example, the names themselves might tell the system what to do with the parameters (e.g., `row-1-col-3-value`), the system might build a database update assuming that the parameter names are database column names, or the servlet might look for a few specific names and then pass the rest of the names to another application.

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. If there are no parameters in the current request, `getParameterNames` returns an empty `Enumeration` (not `null`). Note that `Enumeration` is an interface that merely guarantees that the actual class will have `hasMoreElements` and `nextElement` methods: there is no guarantee that any particular underlying data structure will be used. And, since some common data structures (hash tables, in particular) scramble the order of the elements, you should not count on `getParameterNames` returning the parameters in the order in which they appeared in the HTML form.

Core Warning

Don't count on `getParameterNames` *returning the names in any particular order.*

An alternative to `getParameterNames` is `getParameterMap`. This method returns a `Map`: the parameter names (strings) are the table keys and the parameter values (string arrays as returned by `getParameterNames`) are the table values.

Reading Raw Form Data and Parsing Uploaded Files: getReader or getInputStream

Rather than reading individual form parameters, you can access the query data directly by calling `getReader` or `getInputStream` on the `HttpServletRequest` and then using that stream to parse the raw input. Note, however, that if you read the data in this manner, it is not guaranteed to be available with `getParameter`.

Reading the raw data is a bad idea for regular parameters since the input is neither parsed (separated into entries specific to each parameter) nor URL-decoded (translated so that plus signs become spaces and *%XX* is replaced by the original ASCII or ISO Latin-1 character corresponding to the hex value *XX*). However, reading the raw input is of use in two situations.

The first case in which you might read and parse the data yourself is when the data comes from a custom client rather than by an HTML form. The most common custom client is an applet; applet-servlet communication of this nature is discussed in Volume 2 of this book.

The second situation in which you might read the data yourself is when the data is from an uploaded file. HTML supports a `FORM` element (`<INPUT TYPE="FILE"...>`) that lets the client upload a file to the server. Unfortunately, the servlet API defines no mechanism to read such files. So, you need a third-party library to do so. One of the most popular ones is from the Apache Jakarta Project. See http://jakarta.apache.org/commons/fileupload/ for details.

Reading Input in Multiple Character Sets: setCharacterEncoding

By default, `request.getParameter` interprets input using the server's current character set. To change this default, use the `setCharacterEncoding` method of `ServletRequest`. But, what if input could be in more than one character set? In such a case, you cannot simply call `setCharacterEncoding` with a normal character

set name. The reason for this restriction is that `setCharacterEncoding` must be called *before* you access any request parameters, and in many cases you use a request parameter (e.g., a checkbox) to determine the character set.

So, you are left with two choices: read the parameter in one character set and convert it to another, or use an autodetect feature provided with some character sets.

For the first option, you would read the parameter of interest, use `getBytes` to extract the raw bytes, then pass those bytes to the `String` constructor along with the name of the desired character set. Here is an example that converts a parameter to Japanese:

```
String firstNameWrongEncoding = request.getParameter("firstName");
String firstName =
  new String(firstNameWrongEncoding.getBytes(), "Shift_JIS");
```

For the second option, you would use a character set that supports detection and conversion from the default set. A full list of character sets supported in Java is available at http://java.sun.com/j2se/1.4.1/docs/guide/intl/encoding.doc.html. For example, to allow input in either English or Japanese, you might use the following.

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

4.3 Example: Reading Three Parameters

Listing 4.1 presents a simple servlet called `ThreeParams` that reads form parameters named `param1`, `param2`, and `param3` and places their values in a bulleted list. Although you are required to specify *response* settings (see Chapters 6 and 7) before beginning to generate the content, you are not required to read the *request* parameters at any particular place in your code. So, we read the parameters only when we are ready to use them. Also recall that since the `ThreeParams` class is in the `coreservlets` package, it is deployed to the coreservlets subdirectory of the WEB-INF/classes directory of your Web application (the default Web application in this case).

As we will see later, this servlet is a perfect example of a case that would be dramatically simpler with JSP. See Section 11.6 (Comparing Servlets to JSP Pages) for an equivalent JSP version.

Listing 4.1 ThreeParams.java

```
package coreservlets;

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";
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=\"CENTER\">" + title + "</H1>\n" +
                "<UL>\n" +
                "  <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>\n" +
                "</BODY></HTML>");
  }
}
```

Listing 4.2 shows an HTML form that collects user input and sends it to this servlet. By using an ACTION URL beginning with a slash (/servlet/coreservlets.ThreeParams), you can install the form anywhere in the default Web application; you can move the HTML form to another directory or move both the HTML form and the servlet to another machine, all without editing the HTML form or the servlet. The general principle that form URLs beginning with slashes increases portability holds true even when you use custom Web applications, but you have to include the Web application

prefix in the URL. See Section 2.11 (Web Applications: A Preview) for details on Web applications. There are other ways to write the URLs that also simplify portability, but the most important point is to use relative URLs (no host name), not absolute ones (i.e., http://*host*/...). If you use absolute URLs, you have to edit the forms whenever you move the Web application from one machine to another. Since you almost certainly develop on one machine and deploy on another, use of absolute URLs should be strictly avoided.

Core Approach

Use form `ACTION` *URLs that are relative, not absolute.*

Listing 4.2 ThreeParamsForm.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD><TITLE>Collecting Three Parameters</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN="CENTER">Collecting Three Parameters</H1>

<FORM ACTION="/servlet/coreservlets.ThreeParams">
  First Parameter:  <INPUT TYPE="TEXT" NAME="param1"><BR>
  Second Parameter: <INPUT TYPE="TEXT" NAME="param2"><BR>
  Third Parameter:  <INPUT TYPE="TEXT" NAME="param3"><BR>
  <CENTER><INPUT TYPE="SUBMIT"></CENTER>
</FORM>

</BODY></HTML>
```

Recall that the location of the default Web application varies from server to server. HTML forms go in the top-level directory or in subdirectories other than WEB-INF. If we place the HTML page in the form-data subdirectory and access it from the local machine, then the full installation location on the three sample servers used in the book is as follows:

- **Tomcat Location**
  install_dir/webapps/ROOT/form-data/ThreeParamsForm.html
- **JRun Location**
  install_dir/servers/default/default-ear/default-war/form-data/ThreeParamsForm.html

- **Resin Location**
  install_dir/doc/form-data/ThreeParamsForm.html
- **Corresponding URL**
  http://localhost/form-data/ThreeParamsForm.html

Figure 4–1 shows the HTML form when the user has entered the home directory names of three famous Internet personalities. OK, OK, only two of them are famous,[1] but the point here is that the tilde (~) is a nonalphanumeric character and will be URL-encoded by the browser when the form is submitted. Figure 4–2 shows the result of the servlet; note the URL-encoded values on the address line but the original form field values in the output: `getParameter` always returns the values as the end user typed them in, regardless of how they were sent over the network.

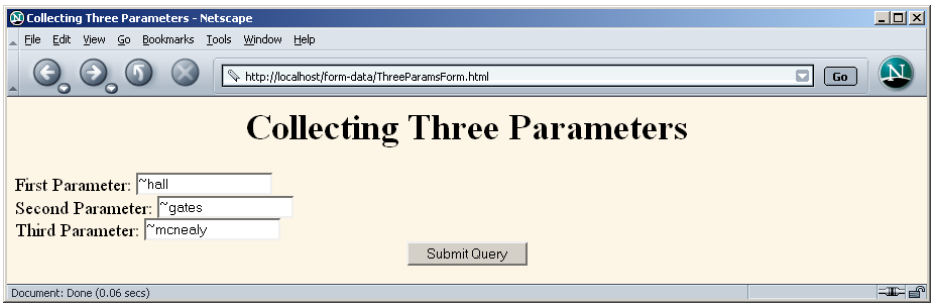


Figure 4–1 Front end to parameter-processing servlet.

Figure 4–2 Result of parameter-processing servlet: request parameters are URL-decoded automatically.

1. Gates isn't *that* famous, after all.

4.4 Example: Reading All Parameters

The previous example extracts parameter values from the form data according to prespecified parameter names. It also assumes that each parameter has exactly one value. Here's an example that looks up *all* the parameter names that are sent and puts their values in a table. It highlights parameters that have missing values as well as ones that have multiple values. Although this approach is rarely used in production servlets (if you don't know the names of the form parameters, you probably don't know what to do with them), it is quite useful for debugging.

First, the servlet looks up all the parameter names with the `getParameterNames` method of `HttpServletRequest`. This method returns an `Enumeration` that contains the parameter names in an unspecified order. Next, the servlet loops down the `Enumeration` in the standard manner, using `hasMoreElements` to determine when to stop and using `nextElement` to get each parameter name. Since `nextElement` returns an `Object`, the servlet casts the result to a `String` and passes that to `getParameterValues`, yielding an array of strings. If that array is one entry long and contains only an empty string, then the parameter had no values and the servlet generates an italicized "No Value" entry. If the array is more than one entry long, then the parameter had multiple values and the values are displayed in a bulleted list. Otherwise, the single value is placed directly into the table.

The source code for the servlet is shown in Listing 4.3; Listing 4.4 shows the HTML code for a front end you can use to try out the servlet. Figures 4–3 and 4–4 show the result of the HTML front end and the servlet, respectively.

Listing 4.3 ShowParameters.java

```
package coreservlets;

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

/** Shows all the parameters sent to the servlet via either
 *  GET or POST. Specially marks parameters that have
 *  no values or multiple values.
 */
```

Listing 4.3 ShowParameters.java *(continued)*

```
public class ShowParameters 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";
    String title = "Reading All Request Parameters";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                "<TABLE BORDER=1 ALIGN=CENTER>\n" +
                "<TR BGCOLOR=\"#FFAD00\">\n" +
                "<TH>Parameter Name<TH>Parameter Value(s)");
    Enumeration paramNames = request.getParameterNames();
    while(paramNames.hasMoreElements()) {
      String paramName = (String)paramNames.nextElement();
      out.print("<TR><TD>" + paramName + "\n<TD>");
      String[] paramValues =
        request.getParameterValues(paramName);
      if (paramValues.length == 1) {
        String paramValue = paramValues[0];
        if (paramValue.length() == 0)
          out.println("<I>No Value</I>");
        else
          out.println(paramValue);
      } else {
        out.println("<UL>");
        for(int i=0; i<paramValues.length; i++) {
          out.println("<LI>" + paramValues[i]);
        }
        out.println("</UL>");
      }
    }
    out.println("</TABLE>\n</BODY></HTML>");
  }

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

Notice that the servlet uses a `doPost` method that simply calls `doGet`. That's because we want it to be able to handle *both* `GET` and `POST` requests. This approach is a good standard practice if you want HTML interfaces to have some flexibility in how they send data to the servlet. See the discussion of the `service` method in Section 3.6 (The Servlet Life Cycle) for a discussion of why having `doPost` call `doGet` (or vice versa) is preferable to overriding `service` directly. The HTML form from Listing 4.4 uses `POST`, as should *all* forms that have password fields (for details, see Chapter 19, "Creating and Processing HTML Forms"). However, the `ShowParameters` servlet is not specific to that particular front end, so the source code archive site at http://www.coreservlets.com/ includes a similar HTML form that uses `GET` for you to experiment with.

Listing 4.4 ShowParametersPostForm.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD><TITLE>A Sample FORM using POST</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN="CENTER">A Sample FORM using POST</H1>
<FORM ACTION="/servlet/coreservlets.ShowParameters"
      METHOD="POST">
  Item Number: <INPUT TYPE="TEXT" NAME="itemNum"><BR>
  Description: <INPUT TYPE="TEXT" NAME="description"><BR>
  Price Each: <INPUT TYPE="TEXT" NAME="price" VALUE="$"><BR>
  <HR>
  First Name: <INPUT TYPE="TEXT" NAME="firstName"><BR>
  Last Name: <INPUT TYPE="TEXT" NAME="lastName"><BR>
  Middle Initial: <INPUT TYPE="TEXT" NAME="initial"><BR>
  Shipping Address:
  <TEXTAREA NAME="address" ROWS=3 COLS=40></TEXTAREA><BR>
  Credit Card:<BR>
    <INPUT TYPE="RADIO" NAME="cardType"
                     VALUE="Visa">Visa<BR>
    <INPUT TYPE="RADIO" NAME="cardType"
                     VALUE="MasterCard">MasterCard<BR>
    <INPUT TYPE="RADIO" NAME="cardType"
                     VALUE="Amex">American Express<BR>
    <INPUT TYPE="RADIO" NAME="cardType"
                     VALUE="Discover">Discover<BR>
    <INPUT TYPE="RADIO" NAME="cardType"
                     VALUE="Java SmartCard">Java SmartCard<BR>
```

Listing 4.4 ShowParametersPostForm.html *(continued)*

```
  Credit Card Number:
  <INPUT TYPE="PASSWORD" NAME="cardNum"><BR>
  Repeat Credit Card Number:
  <INPUT TYPE="PASSWORD" NAME="cardNum"><BR><BR>
  <CENTER><INPUT TYPE="SUBMIT" VALUE="Submit Order"></CENTER>
</FORM>
</BODY></HTML>
```

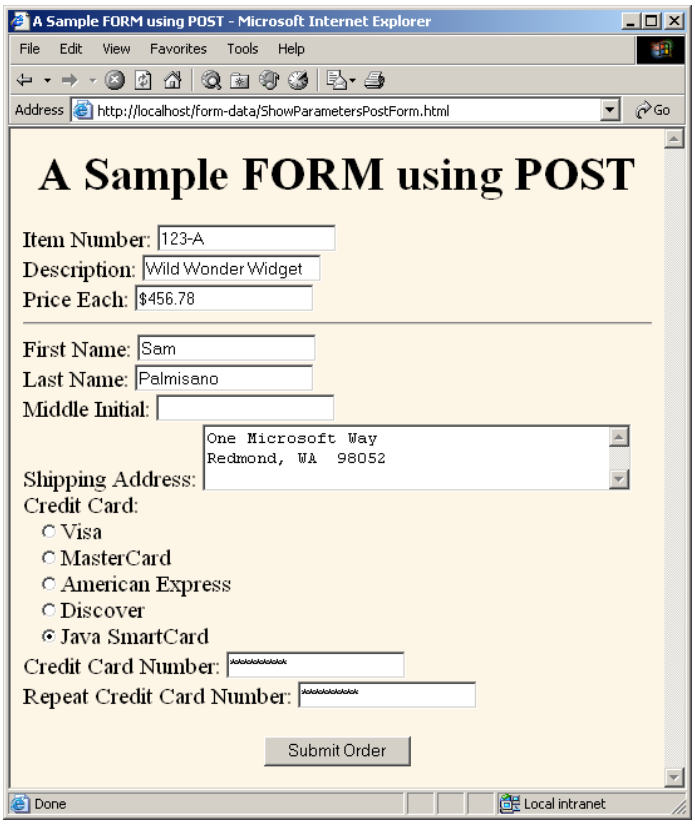


Figure 4–3 HTML form that collects data for the `ShowParameters` servlet.

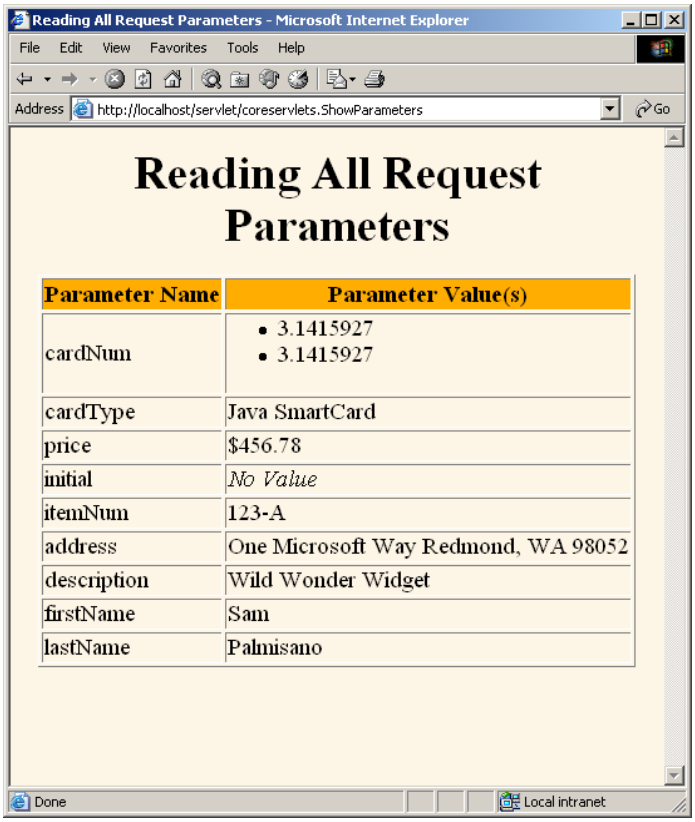
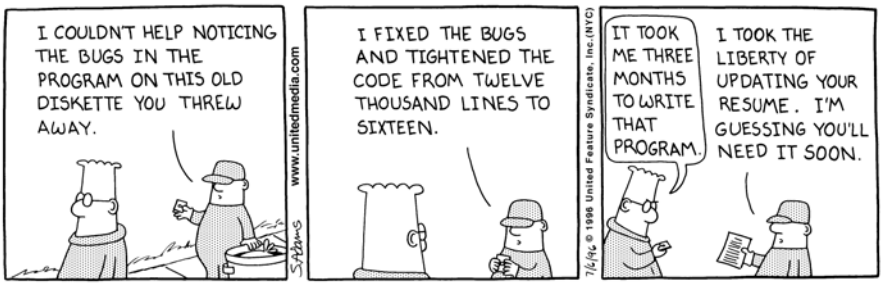


Figure 4–4 Result of the `ShowParameters` servlet.

4.5 Using Default Values When Parameters Are Missing or Malformed

Online job services have become increasingly popular of late. A reputable site provides a useful service to job seekers by giving their skills wide exposure and provides a useful service to employers by giving them access to a large pool of prospective employees. This section presents a servlet that handles part of such a site: the creation of online résumés from user-submitted data. Now, the question is: what should the servlet do when the user fails to supply the necessary information? This question has two answers: use default values or redisplay the form (prompting the user for missing values). This section illustrates the use of default values; Section 4.8 illustrates redisplay of the form.



DILBERT reprinted by permission of United Feature Syndicate, Inc.

When examining request parameters, you need to check for three conditions:

1. **The value is `null`.** A call to `request.getParameter` returns `null` if the form contains no textfield or other element of the expected name so that the parameter name does not appear in the request at all. This can happen when the end user uses an incorrect HTML form or when a bookmarked URL containing `GET` data is used but the parameter names have changed since the URL was bookmarked. To avoid a `NullPointerException`, you have to check for `null` *before* you try to call any methods on the string that results from `getParameter`.
2. **The value is an empty string.** A call to `request.getParameter` returns an empty string (i.e., `""`) if the associated textfield is empty when the form is submitted. To check for an empty string, compare the string to `""` by using `equals` or compare the length of the string to 0. *Do not use the == operator*; in the Java programming language, `==` always tests whether the two arguments are the same object (at the same memory location), not whether the two objects look similar. Just to be safe, it is also a good idea to call `trim` to remove any white space that the user may have entered, since in most scenarios you want to treat pure white space as missing data. So, for example, a test for missing values might look like the following.

   ```
   String param = request.getParameter("someName");
   if ((param == null) || (param.trim().equals(""))) {
     doSomethingForMissingValues(...);
   } else {
     doSomethingWithParameter(param);
   }
   ```

3. **The value is a nonempty string of the wrong format.** What defines the wrong format is application specific: you might expect certain textfields to contain only numeric values, others to have exactly seven characters, and others to only contain single letters.

Note that the use of JavaScript for client-side validation does not remove the need for also doing this type of checking on the server. After all, you are responsible for the server-side application, and often another developer or group is responsible for the forms. You do not want *your* application to crash if *they* fail to detect every type of illegal input. Besides, clients can use their own HTML forms, can manually edit URLs that contain `GET` data, and can disable JavaScript.

Core Approach

Design your servlets to gracefully handle parameters that are missing (`null` or empty string) or improperly formatted. Test your servlets with missing and malformed data as well as with data in the expected format.

Listing 4.5 and Figure 4–5 show the HTML form that acts as the front end to the résumé-processing servlet. If you are not familiar with HTML forms, see Chapter 19. The form uses `POST` to submit the data and it gathers values for various parameter names. The important thing to understand here is what the servlet does with missing and malformed data. This process is summarized in the following list. Listing 4.6 shows the complete servlet code.

- **`name`**, **`title`**, **`email`**, **`languages`**, **`skills`**
  These parameters specify various parts of the résumé. Missing values should be replaced by default values specific to the parameter. The servlet uses a method called `replaceIfMissing` to accomplish this task.
- **`fgColor`**, **`bgColor`**
  These parameters give the colors of the foreground and background of the page. Missing values should result in black for the foreground and white for the background. The servlet again uses `replaceIfMissing` to accomplish this task.
- **`headingFont`**, **`bodyFont`**
  These parameters designate the font to use for headings and the main text, respectively. A missing value or a value of "default" should result in a sans-serif font such as Arial or Helvetica. The servlet uses a method called `replaceIfMissingOrDefault` to accomplish this task.
- **`headingSize`**, **`bodySize`**
  These parameters specify the point size for main headings and body text, respectively. Subheadings will be displayed in a slightly smaller size than the main headings. Missing values or nonnumeric values should result in a default size (32 for headings, 18 for body). The servlet uses a call to `Integer.parseInt` and a `try/catch` block for `NumberFormatException` to handle this case.

Listing 4.5 SubmitResume.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD><TITLE>Free Resume Posting</TITLE>
  <LINK REL=STYLESHEET
        HREF="jobs-site-styles.css"
        TYPE="text/css">
</HEAD>
<BODY>
<H1>hot-computer-jobs.com</H1>
<P CLASS="LARGER">
To use our <I>free</I> resume-posting service, simply fill
out the brief summary of your skills below. Use "Preview"
to check the results, then press "Submit" once it is
ready. Your mini-resume will appear online within 24 hours.</P>
<HR>
<FORM ACTION="/servlet/coreservlets.SubmitResume"
      METHOD="POST">
<DL>
<DT><B>First, give some general information about the look of
your resume:</B>
<DD>Heading font:
    <INPUT TYPE="TEXT" NAME="headingFont" VALUE="default">
<DD>Heading text size:
    <INPUT TYPE="TEXT" NAME="headingSize" VALUE=32>
<DD>Body font:
    <INPUT TYPE="TEXT" NAME="bodyFont" VALUE="default">
<DD>Body text size:
    <INPUT TYPE="TEXT" NAME="bodySize" VALUE=18>
<DD>Foreground color:
    <INPUT TYPE="TEXT" NAME="fgColor" VALUE="BLACK">
<DD>Background color:
    <INPUT TYPE="TEXT" NAME="bgColor" VALUE="WHITE">

<DT><B>Next, give some general information about yourself:</B>
<DD>Name: <INPUT TYPE="TEXT" NAME="name">
<DD>Current or most recent title:
    <INPUT TYPE="TEXT" NAME="title">
<DD>Email address: <INPUT TYPE="TEXT" NAME="email">
<DD>Programming Languages:
    <INPUT TYPE="TEXT" NAME="languages">

<DT><B>Finally, enter a brief summary of your skills and
    experience:</B> (use &lt;P&gt; to separate paragraphs.
    Other HTML markup is also permitted.)
<DD><TEXTAREA NAME="skills"
              ROWS=10 COLS=60 WRAP="SOFT"></TEXTAREA>
```

Listing 4.5 SubmitResume.html *(continued)*

```
</DL>
  <CENTER>
    <INPUT TYPE="SUBMIT" NAME="previewButton" Value="Preview">
    <INPUT TYPE="SUBMIT" NAME="submitButton" Value="Submit">
  </CENTER>
</FORM>
<HR>
<P CLASS="TINY">See our privacy policy
<A HREF="we-will-spam-you.html">here</A>.</P>
</BODY></HTML>
```

Figure 4–5 Front end to résumé-previewing servlet.

Listing 4.6 SubmitResume.java

```
package coreservlets;

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

/** Servlet that handles previewing and storing resumes
 *  submitted by job applicants.
 */

public class SubmitResume extends HttpServlet {
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    if (request.getParameter("previewButton") != null) {
      showPreview(request, out);
    } else {
      storeResume(request);
      showConfirmation(request, out);
    }
  }

  /** Shows a preview of the submitted resume. Takes
   *  the font information and builds an HTML
   *  style sheet out of it, then takes the real
   *  resume information and presents it formatted with
   *  that style sheet.
   */

  private void showPreview(HttpServletRequest request,
                           PrintWriter out) {
    String headingFont = request.getParameter("headingFont");
    headingFont = replaceIfMissingOrDefault(headingFont, "");
    int headingSize =
      getSize(request.getParameter("headingSize"), 32);
    String bodyFont = request.getParameter("bodyFont");
    bodyFont = replaceIfMissingOrDefault(bodyFont, "");
    int bodySize =
      getSize(request.getParameter("bodySize"), 18);
    String fgColor = request.getParameter("fgColor");
    fgColor = replaceIfMissing(fgColor, "BLACK");
    String bgColor = request.getParameter("bgColor");
```

Listing 4.6 SubmitResume.java *(continued)*

```
  bgColor = replaceIfMissing(bgColor, "WHITE");
  String name = request.getParameter("name");
  name = replaceIfMissing(name, "Lou Zer");
  String title = request.getParameter("title");
  title = replaceIfMissing(title, "Loser");
  String email = request.getParameter("email");
  email =
    replaceIfMissing(email, "contact@hot-computer-jobs.com");
  String languages = request.getParameter("languages");
  languages = replaceIfMissing(languages, "<I>None</I>");
  String languageList = makeList(languages);
  String skills = request.getParameter("skills");
  skills = replaceIfMissing(skills, "Not many, obviously.");
  out.println
    (ServletUtilities.DOCTYPE + "\n" +
     "<HTML><HEAD><TITLE>Resume for " + name + "</TITLE>\n" +
     makeStyleSheet(headingFont, headingSize,
                    bodyFont, bodySize,
                    fgColor, bgColor) + "\n" +
     "</HEAD>\n" +
     "<BODY>\n" +
     "<CENTER>\n"+
     "<SPAN CLASS=\"HEADING1\">" + name + "</SPAN><BR>\n" +
     "<SPAN CLASS=\"HEADING2\">" + title + "<BR>\n" +
     "<A HREF=\"mailto:" + email + "\">" + email +
         "</A></SPAN>\n" +
     "</CENTER><BR><BR>\n" +
     "<SPAN CLASS=\"HEADING3\">Programming Languages" +
     "</SPAN>\n" +
     makeList(languages) + "<BR><BR>\n" +
     "<SPAN CLASS=\"HEADING3\">Skills and Experience" +
     "</SPAN><BR><BR>\n" +
     skills + "\n" +
     "</BODY></HTML>");
}

/** Builds a cascading style sheet with information
 *  on three levels of headings and overall
 *  foreground and background cover. Also tells
 *  Internet Explorer to change color of mailto link
 *  when mouse moves over it.
 */
```

Listing 4.6 SubmitResume.java *(continued)*

```
private String makeStyleSheet(String headingFont,
                              int heading1Size,
                              String bodyFont,
                              int bodySize,
                              String fgColor,
                              String bgColor) {
  int heading2Size = heading1Size*7/10;
  int heading3Size = heading1Size*6/10;
  String styleSheet =
    "<STYLE TYPE=\"text/css\">\n" +
    "<!--\n" +
    ".HEADING1 { font-size: " + heading1Size + "px;\n" +
    "            font-weight: bold;\n" +
    "            font-family: " + headingFont +
                  "Arial, Helvetica, sans-serif;\n" +
    "}\n" +
    ".HEADING2 { font-size: " + heading2Size + "px;\n" +
    "            font-weight: bold;\n" +
    "            font-family: " + headingFont +
                  "Arial, Helvetica, sans-serif;\n" +
    "}\n" +
    ".HEADING3 { font-size: " + heading3Size + "px;\n" +
    "            font-weight: bold;\n" +
    "            font-family: " + headingFont +
                  "Arial, Helvetica, sans-serif;\n" +
    "}\n" +
    "BODY { color: " + fgColor + ";\n" +
    "       background-color: " + bgColor + ";\n" +
    "       font-size: " + bodySize + "px;\n" +
    "       font-family: " + bodyFont +
                  "Times New Roman, Times, serif;\n" +
    "}\n" +
    "A:hover { color: red; }\n" +
    "-->\n" +
    "</STYLE>";
  return(styleSheet);
}

/** Replaces null strings (no such parameter name) or
 *  empty strings (e.g., if textfield was blank) with
 *  the replacement. Returns the original string otherwise.
 */
```

Listing 4.6 SubmitResume.java *(continued)*

```
private String replaceIfMissing(String orig,
                                 String replacement) {
  if ((orig == null) || (orig.trim().equals(""))) {
    return(replacement);
  } else {
    return(orig);
  }
}

// Replaces null strings, empty strings, or the string
// "default" with the replacement.
// Returns the original string otherwise.

private String replaceIfMissingOrDefault(String orig,
                                         String replacement) {
  if ((orig == null) ||
      (orig.trim().equals("")) ||
      (orig.equals("default"))) {
    return(replacement);
  } else {
    return(orig + ", ");
  }
}

// Takes a string representing an integer and returns it
// as an int. Returns a default if the string is null
// or in an illegal format.

private int getSize(String sizeString, int defaultSize) {
  try {
    return(Integer.parseInt(sizeString));
  } catch(NumberFormatException nfe) {
    return(defaultSize);
  }
}

// Given "Java,C++,Lisp", "Java C++ Lisp" or
// "Java, C++, Lisp", returns
// "<UL>
//   <LI>Java
//   <LI>C++
//   <LI>Lisp
//  </UL>"
```

Listing 4.6 SubmitResume.java *(continued)*

```
  private String makeList(String listItems) {
    StringTokenizer tokenizer =
      new StringTokenizer(listItems, ", ");
    String list = "<UL>\n";
    while(tokenizer.hasMoreTokens()) {
      list = list + "  <LI>" + tokenizer.nextToken() + "\n";
    }
    list = list + "</UL>";
    return(list);
  }

  /** Shows a confirmation page when the user clicks the
   *  "Submit" button.
   */

  private void showConfirmation(HttpServletRequest request,
                                PrintWriter out) {
    String title = "Submission Confirmed.";
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY>\n" +
                "<H1>" + title + "</H1>\n" +
                "Your resume should appear online within\n" +
                "24 hours. If it doesn't, try submitting\n" +
                "again with a different email address.\n" +
                "</BODY></HTML>");
  }

  /** Why it is bad to give your email address to
   *  untrusted sites.
   */

  private void storeResume(HttpServletRequest request) {
    String email = request.getParameter("email");
    putInSpamList(email);
  }

  private void putInSpamList(String emailAddress) {
    // Code removed to protect the guilty.
  }
}
```

Once the servlet has meaningful values for each of the font and color parameters, it builds a cascading style sheet out of them. Style sheets are a standard way of specifying the font faces, font sizes, colors, indentation, and other formatting information in an HTML 4.0 Web page. Style sheets are usually placed in a separate file so that several

Web pages at a site can share the same style sheet, but in this case it is more convenient to embed the style information directly in the page by use of the `STYLE` element. For more information on style sheets, see http://www.w3.org/TR/REC-CSS1.

After creating the style sheet, the servlet places the job applicant's name, job title, and email address centered under each other at the top of the page. The heading font is used for these lines, and the email address is placed inside a `mailto:` hypertext link so that prospective employers can contact the applicant directly by clicking on the address. The programming languages specified in the `languages` parameter are parsed by `StringTokenizer` (assuming spaces or commas are used to separate the language names) and placed in a bulleted list beneath a "Programming Languages" heading. Finally, the text from the `skills` parameter is placed at the bottom of the page beneath a "Skills and Experience" heading.

Figures 4–6 and 4–7 show results when the required data is supplied and omitted, respectively. Figure 4–8 shows the result of clicking Submit instead of Preview.

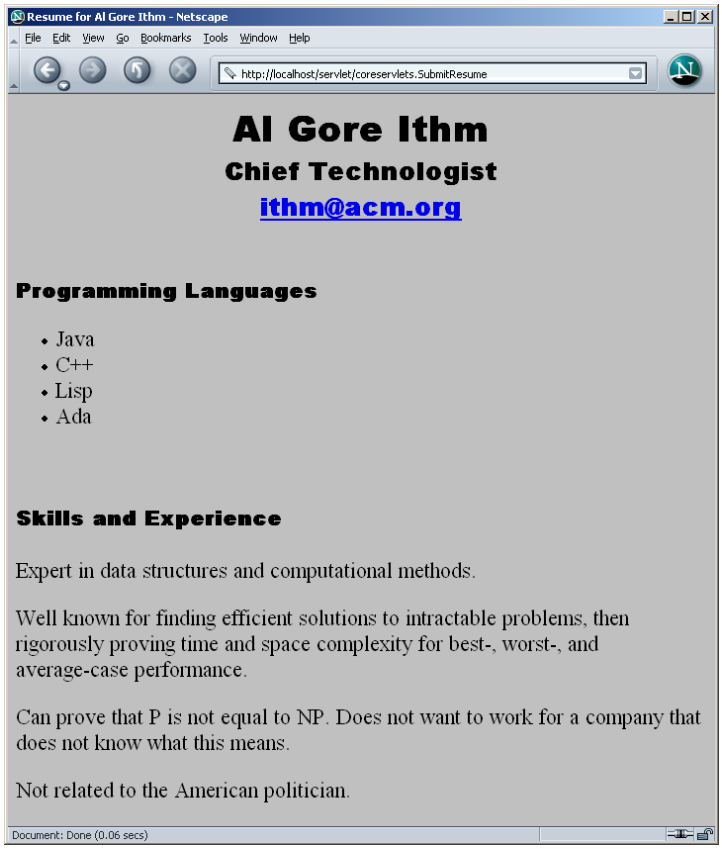


Figure 4–6 Preview of a résumé submission that contained the required form data.

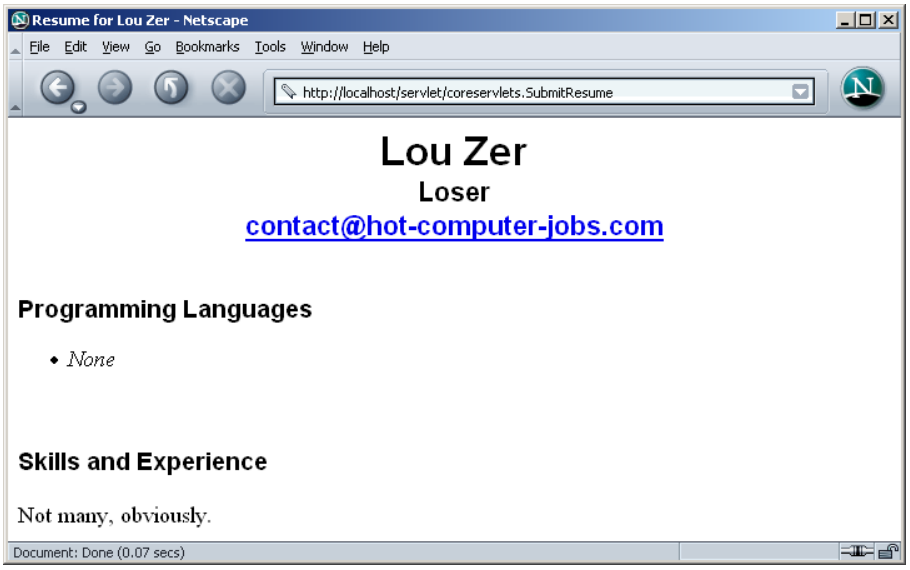


Figure 4–7 Preview of a submission that was missing much of the required data: default values replace the omitted values.

Figure 4–8 Result of submitting the résumé to the database.

4.6 Filtering Strings for HTML-Specific Characters

Normally, when a servlet wants to generate HTML that will contain characters like < or >, it simply uses `<` or `>`, the standard HTML character entities. Similarly, if a servlet wants a double quote or an ampersand to appear inside an HTML attribute value, it uses `&quot;` or `&`. Failing to make these substitutions results in malformed HTML code, since < or > will often be interpreted as part of an HTML markup tag, a double quote in an attribute value may be interpreted as the end of the value, and ampersands are just plain illegal in attribute values. In most cases, it is easy to note the special characters and use the standard HTML replacements. However, there are two cases in which it is not so easy to make this substitution manually.

The first case in which manual conversion is difficult occurs when the string is derived from a program excerpt or another source in which it is already in some standard format. Going through manually and changing all the special characters can be tedious in such a case, but forgetting to convert even one special character can result in your Web page having missing or improperly formatted sections.

The second case in which manual conversion fails is when the string is derived from HTML form data. Here, the conversion absolutely must be performed at runtime, since of course the query data is not known at compile time. If the user accidentally or deliberately enters HTML tags, the generated Web page will contain spurious HTML tags and can have completely unpredictable results (the HTML specification tells browsers what to do with legal HTML; it says nothing about what they should do with HTML containing illegal syntax).

Core Approach

If you read request parameters and display their values in the resultant page, you should filter out the special HTML characters. Failing to do so can result in output that has missing or oddly formatted sections.

Failing to do this filtering for externally accessible Web pages also lets your page become a vehicle for the *cross-site scripting attack*. Here, a malicious programmer embeds `GET` parameters in a URL that refers to one of your servlets (or any other server-side program). These `GET` parameters expand to HTML tags (usually `<SCRIPT>` elements) that exploit known browser bugs. So, an attacker could embed the code in a URL that refers to your site and distribute only the URL, not the malicious Web page itself. That way, the attacker can remain undiscovered more easily and can also exploit

trusted relationships to make users think the scripts are coming from a trusted source (your organization). For more details on this issue, see http://www.cert.org/advisories/CA-2000-02.html and http://www.microsoft.com/technet/security/topics/ExSumCS.asp.

Code for Filtering

Replacing <, >, ", and & in strings is a simple matter, and a number of different approaches can accomplish the task. However, it is important to remember that Java strings are immutable (i.e., can't be modified), so repeated string concatenation involves copying and then discarding many string segments. For example, consider the following two lines:

```
String s1 = "Hello";
String s2 = s1 + " World";
```

Since `s1` cannot be modified, the second line makes a copy of `s1` and appends `"World"` to the copy, then the copy is discarded. To avoid the expense of generating and copying these temporary objects, whenever you perform repeated concatenation within a loop, you should use a mutable data structure; and `StringBuffer` is the natural choice.

Core Approach

If you do string concatenation from within a loop, use `StringBuffer`, not `String`.

Listing 4.7 shows a static `filter` method that uses a `StringBuffer` to efficiently copy characters from an input string to a filtered version, replacing the four special characters along the way.

Listing 4.7 ServletUtilities.java (Excerpt)

```
package coreservlets;
import javax.servlet.*;
import javax.servlet.http.*;

public class ServletUtilities {
  ...

  /** Replaces characters that have special HTML meanings
   *  with their corresponding HTML character entities.
   */
```

Listing 4.7 ServletUtilities.java (Excerpt) *(continued)*

```
  // Note that Javadoc is not used for the more detailed
  // documentation due to the difficulty of making the
  // special chars readable in both plain text and HTML.
  //
  //  Given a string, this method replaces all occurrences of
  //  '<' with '&lt;', all occurrences of '>' with
  //  '&gt;', and (to handle cases that occur inside attribute
  //  values), all occurrences of double quotes with
  //  '&quot;' and all occurrences of '&' with '&amp;'.
  //  Without such filtering, an arbitrary string
  //  could not safely be inserted in a Web page.

  public static String filter(String input) {
    if (!hasSpecialChars(input)) {
      return(input);
    }
    StringBuffer filtered = new StringBuffer(input.length());
    char c;
    for(int i=0; i<input.length(); i++) {
      c = input.charAt(i);
      switch(c) {
        case '<': filtered.append("&lt;"); break;
        case '>': filtered.append("&gt;"); break;
        case '"': filtered.append("&quot;"); break;
        case '&': filtered.append("&amp;"); break;
        default: filtered.append(c);
      }
    }
    return(filtered.toString());
  }

  private static boolean hasSpecialChars(String input) {
    boolean flag = false;
    if ((input != null) && (input.length() > 0)) {
      char c;
      for(int i=0; i<input.length(); i++) {
        c = input.charAt(i);
        switch(c) {
          case '<': flag = true; break;
          case '>': flag = true; break;
          case '"': flag = true; break;
          case '&': flag = true; break;
        }
      }
    }
    return(flag);
  }
}
```

Example: A Servlet That Displays Code Snippets

As an example, consider the HTML form of Listing 4.8 that gathers a snippet of the Java programming language and sends it to the servlet of Listing 4.9 for display.

Now, when the user enters normal input, the result is fine, as illustrated by Figure 4–9. However, as shown in Figure 4–10, the result can be unpredictable when the input contains special characters like < and >. Different browsers can give different results since the HTML specification doesn't say what to do in this case, but most browsers think that the "<b" in "`if (a<b) {`" starts an HTML tag. But, since the characters after the `b` are unrecognized, browsers ignore them until the next >, which is at the end of the `</PRE>` tag. Thus, not only does most of the code snippet disappear, but the browser does not interpret the `</PRE>` tag, so the text after the code snippet is improperly formatted, with a fixed-width font and line wrapping disabled.

Listing 4.10 shows a servlet that works exactly like the previous one except that it filters the special characters from the request parameter value before displaying it. Listing 4.11 shows an HTML form that sends data to it (except for the `ACTION` URL, this form is identical to that shown in Listing 4.8). Figure 4–11 shows the result of the input that failed for the previous servlet: no problem here.

Listing 4.8 CodeForm1.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD><TITLE>Submit Code Sample</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<CENTER>
<H1 ALIGN="CENTER">Submit Code Sample</H1>
<FORM ACTION="/servlet/coreservlets.BadCodeServlet">
  Code:<BR>
  <TEXTAREA ROWS="6" COLS="40" NAME="code"></TEXTAREA><P>
  <INPUT TYPE="SUBMIT" VALUE="Submit Code">
</FORM>
</CENTER></BODY></HTML>
```

Listing 4.9 BadCodeServlet.java

```
package coreservlets;

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

/** Servlet that reads a code snippet from the request
 *  and displays it inside a PRE tag. Fails to filter
 *  the special HTML characters.
 */

public class BadCodeServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Code Sample";
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=\"CENTER\">" + title + "</H1>\n" +
                "<PRE>\n" +
                getCode(request) +
                "</PRE>\n" +
                "Now, wasn't that an interesting sample\n" +
                "of code?\n" +
                "</BODY></HTML>");
  }

  protected String getCode(HttpServletRequest request) {
    return(request.getParameter("code"));
  }
}
```

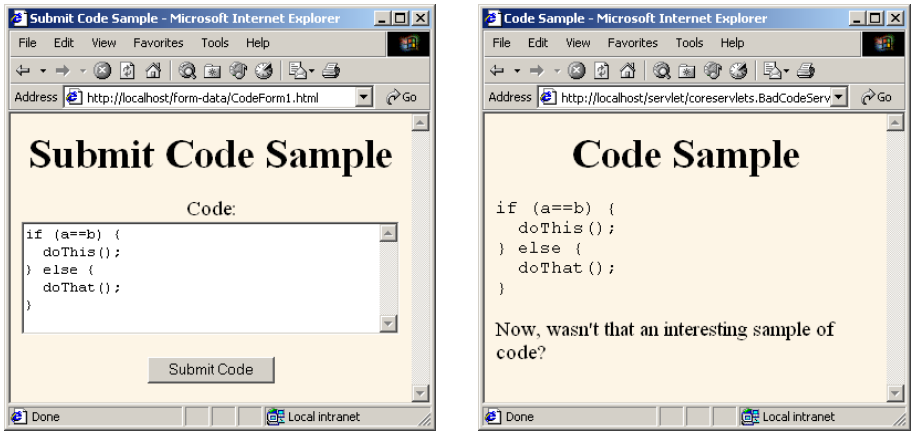


Figure 4–9 `BadCodeServlet`: result is fine when request parameters contain no special characters.

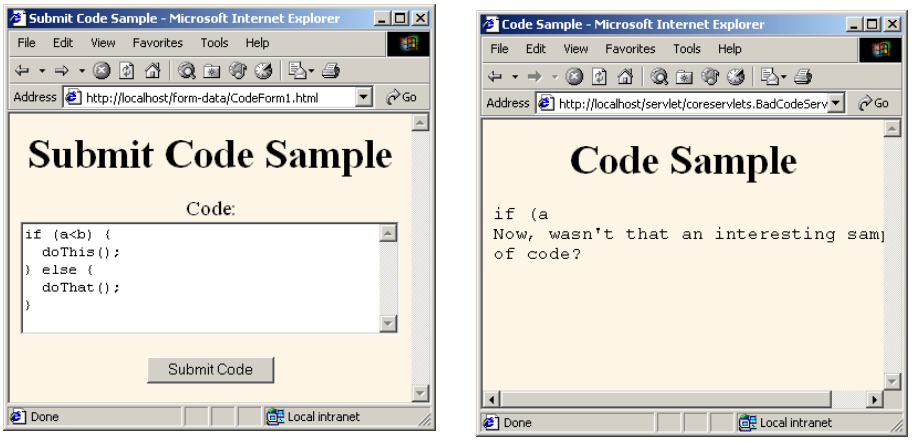


Figure 4–10 `BadCodeServlet`: result has missing and incorrectly formatted sections when request parameters contain special characters.

Listing 4.10 GoodCodeServlet.java

```
package coreservlets;

import javax.servlet.http.*;

/** Servlet that reads a code snippet from the request and displays
 *  it inside a PRE tag. Filters the special HTML characters.
 */

public class GoodCodeServlet extends BadCodeServlet {
  protected String getCode(HttpServletRequest request) {
    return(ServletUtilities.filter(super.getCode(request)));
  }
}
```

Listing 4.11 CodeForm2.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD><TITLE>Submit Code Sample</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<CENTER>
<H1 ALIGN="CENTER">Submit Code Sample</H1>
<FORM ACTION="/servlet/coreservlets.GoodCodeServlet">
  Code:<BR>
  <TEXTAREA ROWS="6" COLS="40" NAME="code"></TEXTAREA><P>
  <INPUT TYPE="SUBMIT" VALUE="Submit Code">
</FORM>
</CENTER></BODY></HTML>
```

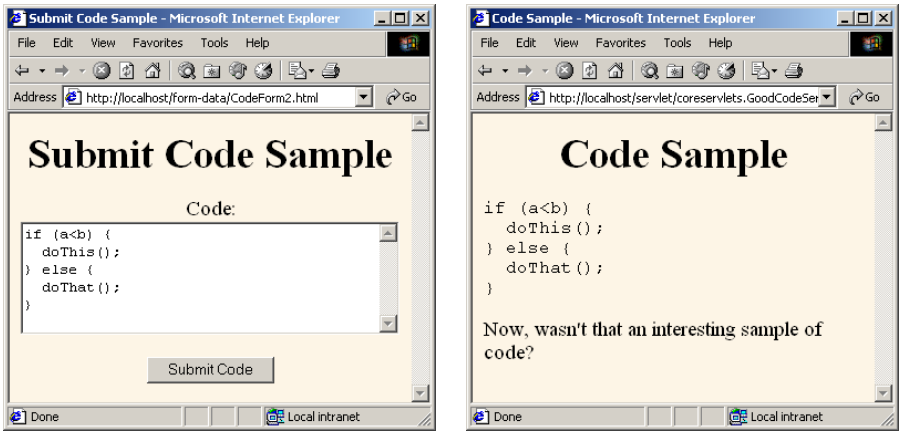


Figure 4–11 `GoodCodeServlet`: result is fine even when request parameters contain special characters.

4.7 Automatically Populating Java Objects from Request Parameters: Form Beans

The `getParameter` method makes it easy to read incoming request parameters: you use the same method from `doGet` as from `doPost`, and the value returned is automatically URL-decoded (i.e., in the format that the end user typed it in, not necessarily in the format in which it was sent over the network). Since the return value of `getParameter` is `String`, however, you have to parse the value yourself (checking for missing or malformed data, of course) if you want other types of values. For example, if you expect `int` or `double` values, you have to pass the result of `getParameter` to `Integer.parseInt` or `Double.parseDouble` and enclose the code inside a `try/catch` block that looks for `NumberFormatException`. If you have many request parameters, this procedure can be quite tedious.

For example, suppose that you have a data object with three `String` fields, two `int` fields, two `double` fields, and a `boolean`. Filling in the object based on a form submission would require eight separate calls to `getParameter`, two calls each to `Integer.parseInt` and `Double.parseDouble`, and some special-purpose code to set the `boolean` flag. It would be nice to do this work automatically.

Now, in JSP, you can use the JavaBeans component architecture to greatly simplify the process of reading request parameters, parsing the values, and storing the results in Java objects. This process is discussed in detail in Chapter 14 (Using JavaBeans Components in JSP Documents). If you are unfamiliar with the idea of beans, refer to that chapter for details. The gist, though, is that an ordinary Java object is considered to be a *bean* if the class uses private fields and has methods that follow the get/set naming convention. The names of the methods (minus the word "get" or "set" and with the first character in lower case) are called *properties*. For example, an arbitrary Java class with a `getName` and `setName` method is said to define a bean that has a property called `name`.

As discussed in Chapter 14, there is special JSP syntax (`property="*"` in a `jsp:setProperty` call) that you can use to populate a bean in one fell swoop. Specifically, this setting indicates that the system should examine all incoming request parameters and pass them to bean properties that match the request parameter name. In particular, if the request parameter is named `param1`, the parameter is passed to the `setParam1` method of the object. Furthermore, simple type conversions are performed automatically. For instance, if there is a request parameter called `numOrdered` and the object has a method called `setNumOrdered` that expects an `int` (i.e., the bean has a `numOrdered` property of type `int`), the `numOrdered` request parameter is automatically converted to an `int` and the resulting value is automatically passed to the `setNumOrdered` method.

Now, if you can do this in JSP, you would think you could do it in servlets as well. After all, as discussed in Chapter 10, JSP pages are really servlets in disguise: each JSP page gets translated into a servlet, and it is the servlet that runs at request time. Furthermore, as we see in Chapter 15 (Integrating Servlets and JSP: The Model View Controller (MVC) Architecture), in complicated scenarios it is often best to combine servlets and JSP pages in such a way that the servlets do the programming work and the JSP pages do the presentation work. So, it is really more important for servlets to be able to read request parameters easily than it is for JSP pages to do so. Surprisingly, however, the servlet specification provides no such capability: the code behind the `property="*"` JSP process is not exposed through a standard API.

Fortunately, the widely used Jakarta Commons package (see http://jakarta.apache.org/commons/) from The Apache Software Foundation contains classes that make it easy to build a utility to automatically associate request parameters with bean properties (i.e., with `setXxx` methods). The next subsection provides information on obtaining the Commons packages, but the important point here is that a static `populateBean` method takes a bean (i.e., a Java object with at least some methods that follow the get/set naming convention) and a `Map` as input and passes all `Map` values to the bean property that matches the associated `Map` key name. This utility also does type conversion automatically, using default values (e.g., 0 for numeric values) instead of throwing exceptions when the corresponding request parameter is malformed. If the bean has no property matching the name, the `Map` entry is ignored; again, no exception is thrown.

Listing 4.12 presents a utility that uses the Jakarta Commons utility to automatically populate a bean according to incoming request parameters. To use it, simply pass the bean and the request object to `BeanUtilities.populateBean`. That's it! You want to put two request parameters into a data object? No problem: one method call is all that's needed. Fifteen request parameters plus type conversion? Same one method call.

Listing 4.12 BeanUtilities.java

```
package coreservlets.beans;

import java.util.*;
import javax.servlet.http.*;
import org.apache.commons.beanutils.BeanUtils;

/** Some utilities to populate beans, usually based on
 *  incoming request parameters. Requires three packages
 *  from the Apache Commons library: beanutils, collections,
 *  and logging. To obtain these packages, see
 *  http://jakarta.apache.org/commons/. Also, the book's
 *  source code archive (see http://www.coreservlets.com/)
```

Listing 4.12 BeanUtilities.java *(continued)*

```
 *  contains links to all URLs mentioned in the book, including
 *  to the specific sections of the Jakarta Commons package.
 *  <P>
 *  Note that this class is in the coreservlets.beans package,
 *  so must be installed in .../coreservlets/beans/.
 */

public class BeanUtilities {
  /** Examines all of the request parameters to see if
   *  any match a bean property (i.e., a setXxx method)
   *  in the object. If so, the request parameter value
   *  is passed to that method. If the method expects
   *  an int, Integer, double, Double, or any of the other
   *  primitive or wrapper types, parsing and conversion
   *  is done automatically. If the request parameter value
   *  is malformed (cannot be converted into the expected
   *  type), numeric properties are assigned zero and boolean
   *  properties are assigned false: no exception is thrown.
   */

  public static void populateBean(Object formBean,
                                  HttpServletRequest request) {
    populateBean(formBean, request.getParameterMap());
  }

  /** Populates a bean based on a Map: Map keys are the
   *  bean property names; Map values are the bean property
   *  values. Type conversion is performed automatically as
   *  described above.
   */

  public static void populateBean(Object bean,
                                  Map propertyMap) {
    try {
      BeanUtils.populate(bean, propertyMap);
    } catch(Exception e) {
      // Empty catch. The two possible exceptions are
      // java.lang.IllegalAccessException and
      // java.lang.reflect.InvocationTargetException.
      // In both cases, just skip the bean operation.
    }
  }
}
```

Putting BeanUtilities to Work

Listing 4.13 shows a servlet that gathers insurance information about an employee, presumably to use it to determine available insurance plans and associated costs. To perform this task, the servlet needs to fill in an insurance information data object (InsuranceInfo.java, Listing 4.14) with information on the employee's name and ID (both of type `String`), number of children (`int`), and whether or not the employee is married (`boolean`). Since this object is represented as a bean, `BeanUtilities.populateBean` can be used to fill in the required information with a single method call. Listing 4.15 shows the HTML form that gathers the data; Figures 4–12 and 4–13 show typical results.

Listing 4.13 SubmitInsuranceInfo.java

```
package coreservlets;

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

/** Example of simplified form processing. Illustrates the
 *  use of BeanUtilities.populateBean to automatically fill
 *  in a bean (Java object with methods that follow the
 *  get/set naming convention) from request parameters.
 */

public class SubmitInsuranceInfo extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    InsuranceInfo info = new InsuranceInfo();
    BeanUtilities.populateBean(info, request);
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    String title = "Insurance Info for " + info.getName();
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<CENTER>\n" +
                "<H1>" + title + "</H1>\n" +
```

Listing 4.13 SubmitInsuranceInfo.java *(continued)*

```
                "<UL>\n" +
                "  <LI>Employee ID: " +
                   info.getEmployeeID() + "\n" +
                "  <LI>Number of children: " +
                   info.getNumChildren() + "\n" +
                "  <LI>Married?: " +
                   info.isMarried() + "\n" +
                "</UL></CENTER></BODY></HTML>");
  }
}
```

Listing 4.14 InsuranceInfo.java

```
package coreservlets.beans;

import coreservlets.*;

/** Simple bean that represents information needed to
 *  calculate an employee's insurance costs. Has String,
 *  int, and boolean properties. Used to demonstrate
 *  automatically filling in bean properties from request
 *  parameters.
 */

public class InsuranceInfo {
  private String name = "No name specified";
  private String employeeID = "No ID specified";
  private int numChildren = 0;
  private boolean isMarried = false;

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

  /** Just in case user enters special HTML characters,
   *  filter them out before storing the name.
   */

  public void setName(String name) {
    this.name = ServletUtilities.filter(name);
  }

  public String getEmployeeID() {
    return(employeeID);
  }
```

Listing 4.14 InsuranceInfo.java *(continued)*

```
  /** Just in case user enters special HTML characters,
   *  filter them out before storing the name.
   */

  public void setEmployeeID(String employeeID) {
    this.employeeID = ServletUtilities.filter(employeeID);
  }

  public int getNumChildren() {
    return(numChildren);
  }

  public void setNumChildren(int numChildren) {
    this.numChildren = numChildren;
  }

  /** Bean convention: name getter method "isXxx" instead
   *  of "getXxx" for boolean methods.
   */

  public boolean isMarried() {
    return(isMarried);
  }

  public void setMarried(boolean isMarried) {
    this.isMarried = isMarried;
  }
}
```

Listing 4.15 InsuranceForm.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD><TITLE>Employee Insurance Signup</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<CENTER>
<H1>Employee Insurance Signup</H1>

<FORM ACTION="/servlet/coreservlets.SubmitInsuranceInfo">
  Name:  <INPUT TYPE="TEXT" NAME="name"><BR>
  Employee ID: <INPUT TYPE="TEXT" NAME="employeeID"><BR>
  Number of Children:  <INPUT TYPE="TEXT" NAME="numChildren"><BR>
  <INPUT TYPE="CHECKBOX" NAME="married" VALUE="true">Married?<BR>
  <CENTER><INPUT TYPE="SUBMIT"></CENTER>
</FORM>

</CENTER></BODY></HTML>
```

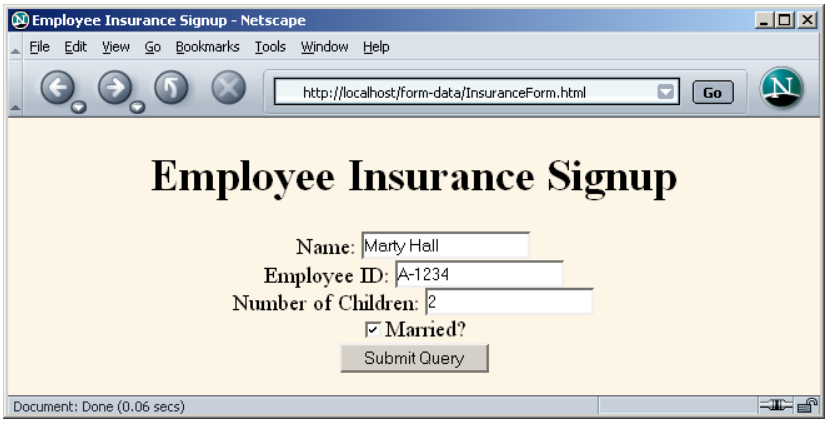


Figure 4–12 Front end to insurance-processing servlet.

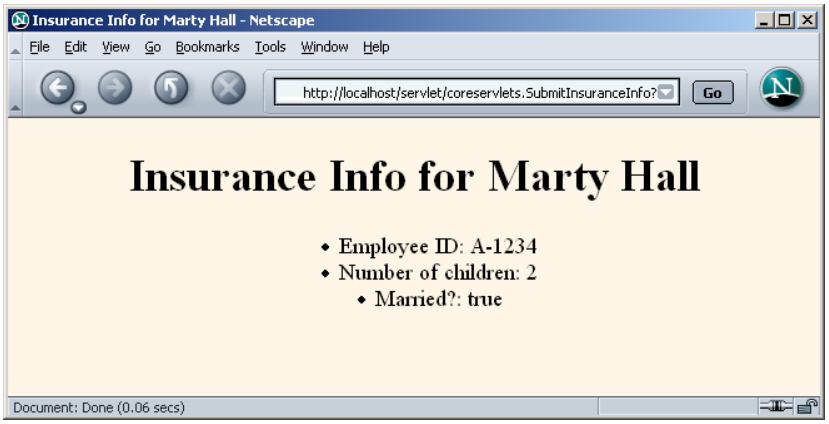


Figure 4–13 Insurance-processing servlet: the gathering of request data is greatly simplified by use of `BeanUtilities.populateBean`.

Obtaining and Installing the Jakarta Commons Packages

Most of the work of our `BeanUtilities` class is done by the Jakarta Commons `BeanUtils` component. This component performs the reflection (determination of what writable bean properties—`set`*Xxx* methods—the object has) and the type conversion (parsing a String as an `int`, `double`, `boolean`, or other primitive or wrapper type). So, `BeanUtilities` will not work unless you install the Jakarta

Commons `BeanUtils`. However, since `BeanUtils` depends on two other Jakarta Commons components—`Collections` and `Logging`—you have to download and install all three.

To download these components, start at http://jakarta.apache.org/commons/, look for the "Components Repository" heading in the left column, and, for each of the three components, download the JAR file for the latest version. (Our code is based on version 1.5 of the `BeanUtils`, but it is likely that any recent version will work identically.) Perhaps the easiest way to download the components is to go to http://www.coreservlets.com/, go to Chapter 4 of the source code archive, and look for the direct links to the three JAR files.

The most portable way to install the components is to follow the standard approach:

- For development, list the three JAR files in your `CLASSPATH`.
- For deployment, put the three JAR files in the WEB-INF/lib directory of your Web application.

When dealing with JAR files like these—used in multiple Web applications—many developers use server-specific features that support sharing of JAR files across Web applications. For example, Tomcat permits common JAR files to be placed in *tomcat_install_dir*/common/lib. Another shortcut that many people use on their development machines is to drop the three JAR files into *sdk_install_dir*/jre/lib/ext. Doing so makes the JAR files automatically accessible both to the development environment and to the locally installed server. These are both useful tricks as long as you remember that *your-web-app*/WEB-INF/lib is the only standardized location on the deployment server.

4.8 Redisplaying the Input Form When Parameters Are Missing or Malformed

It sometimes makes sense to use default values when the user fails to fill in certain form fields. Other times, however, there are no reasonable default values to use, and the form should be redisplayed to the user. Two desirable capabilities make the use of a normal HTML form impossible in this scenario:

- Users should not have to reenter values that they already supplied.
- Missing form fields should be marked prominently.

Redisplay Options

So, what can be done to implement these capabilities? Well, a full description of the possible approaches is a bit complicated and requires knowledge of several techniques (e.g., Struts, JSTL) that are not covered in Volume 1 of this book. So, you should refer to Volume 2 for details, but here is a quick preview.

- **Have the same servlet present the form, process the data, and present the results.** The servlet first looks for incoming request data: if it finds none, it presents a blank form. If the servlet finds partial request data, it extracts the partial data, puts it back into the form, and marks the other fields as missing. If the servlet finds the full complement of required data, it processes the request and displays the results. The form omits the `ACTION` attribute so that form submissions automatically go to the same URL as the form itself. This is the only approach for which we have already covered all of the necessary techniques, so this is the approach illustrated in this section.
- **Have one servlet present the form; have a second servlet process the data and present the results.** This option is better than the first since it divides up the labor and keeps each servlet smaller and more manageable. However, using this approach requires two techniques we have not yet covered: how to transfer control from one servlet to another and how to access user-specific data in one servlet that was created in another. Transferring from one servlet to another can be done with `response.sendRedirect` (see Chapter 6, "Generating the Server Response: HTTP Status Codes") or the `forward` method of `RequestDispatcher` (see Chapter 15, "Integrating Servlets and JSP: The Model View Controller (MVC) Architecture"). The easiest way to pass the data from the processing servlet back to the form-display servlet is to store it in the `HttpSession` object (see Chapter 9, "Session Tracking").
- **Have a JSP page "manually" present the form; have a servlet or JSP page process the data and present the results.** This is an excellent option, and it is widely used. However, it requires knowledge of JSP in addition to knowledge of the two techniques mentioned in the previous bullet (how to transfer the user from the data-processing servlet to the form page and how to use session tracking to store user-specific data). In particular, you need to know how to use JSP expressions (see Chapter 11, "Invoking Java Code with JSP Scripting Elements") or `jsp:getProperty` (see Chapter 14, "Using JavaBeans Components in JSP Documents") to extract the partial data from the data object and put it into the HTML form.

- **Have a JSP page present the form, automatically filling in the fields with values obtained from a data object. Have a servlet or JSP page process the data and present the results.** This is perhaps the best option of all. But, in addition to the techniques described in the previous bullet, it requires custom JSP tags that mimic HTML form elements but use designated values automatically. You can write these tags yourself or you can use ready-made versions such as those that come with JSTL or Apache Struts (see Volume 2 for coverage of custom tags, JSTL, and Struts).

A Servlet That Processes Auction Bids

To illustrate the first of the form-redisplay options, consider a servlet that processes bids at an auction site. Figures 4–14 through 4–16 show the desired outcome: the servlet initially displays a blank form, redisplays the form with missing data marked when partial data is submitted, and processes the request when complete data is submitted.

To accomplish this behavior, the servlet (Listing 4.16) performs the following steps.

1. Fills in a `BidInfo` object (Listing 4.17) from the request data, using `BeanUtilities.populateBean` (see Section 4.7, "Automatically Populating Java Objects from Request Parameters: Form Beans") to automatically match up request parameter names with bean properties and to perform simple type conversion.
2. Checks whether that `BidInfo` object is completely empty (no fields changed from the default). If so, it calls `showEntryForm` to display the initial input form.
3. Checks whether the `BidInfo` object is partially empty (some, but not all, fields changed from the default). If so, it calls `showEntryForm` to display the input form with a warning message and with the missing fields highlighted. Fields in which the user already entered data keep their previous values.
4. Checks whether the `BidInfo` object is completely filled in. If so, it calls `showBid` to process the data and present the result.

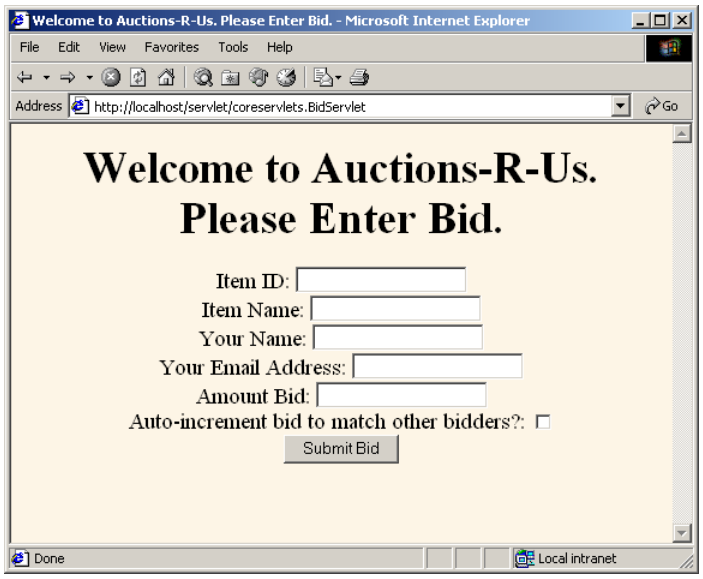


Figure 4–14 Original form of servlet: it presents a form to collect data about a bid at an auction.

Figure 4–15 Bid servlet with incomplete data. If the user submits a form that is not fully filled in, the bid servlet redisplays the form. The user's previous partial data is maintained, and missing fields are highlighted.

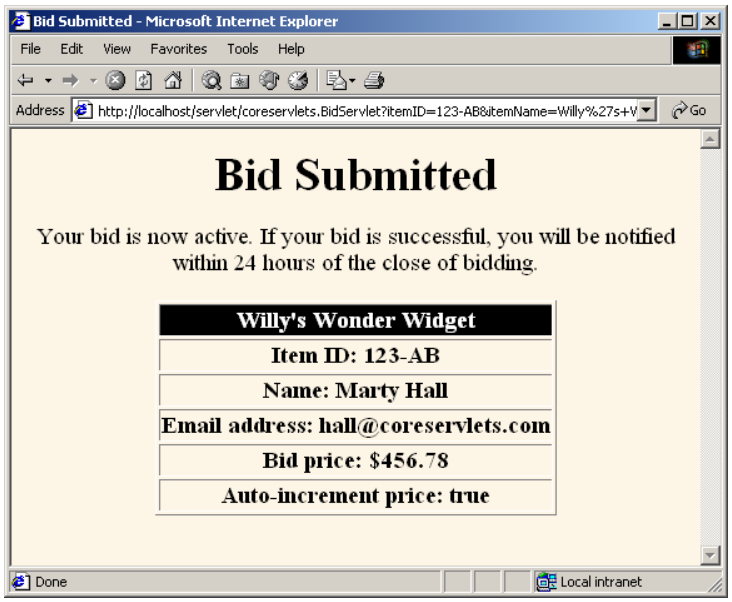


Figure 4–16 Bid servlet with complete data: it presents the results.

Listing 4.16 BidServlet.java

```
package coreservlets;

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

/** Example of simplified form processing. Shows two features:
 *  <OL>
 *    <LI>Automatically filling in a bean based on the
 *        incoming request parameters.
 *    <LI>Using the same servlet both to generate the input
 *        form and to process the results. That way, when
 *        fields are omitted, the servlet can redisplay the
 *        form without making the user reenter previously
 *        entered values.
 *  </UL>
 */
```

Listing 4.16 BidServlet.java *(continued)*

```
public class BidServlet extends HttpServlet {

  /** Try to populate a bean that represents information
   *  in the form data sent by the user. If this data is
   *  complete, show the results. If the form data is
   *  missing or incomplete, display the HTML form
   *  that gathers the data.
   */

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    BidInfo bid = new BidInfo();
    BeanUtilities.populateBean(bid, request);
    if (bid.isComplete()) {
      // All required form data was supplied: show result.
      showBid(request, response, bid);
    } else {
      // Form data was missing or incomplete: redisplay form.
      showEntryForm(request, response, bid);
    }
  }

  /** All required data is present: show the results page. */

  private void showBid(HttpServletRequest request,
                       HttpServletResponse response,
                       BidInfo bid)
      throws ServletException, IOException {
    submitBid(bid);
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Bid Submitted";
    out.println
      (DOCTYPE +
       "<HTML>\n" +
       "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
       "<BODY BGCOLOR=\"#FDF5E6\"><CENTER>\n" +
       "<H1>" + title + "</H1>\n" +
       "Your bid is now active. If your bid is successful,\n" +
       "you will be notified within 24 hours of the close\n" +
       "of bidding.\n" +
       "<P>\n" +
       "<TABLE BORDER=1>\n" +
       "  <TR><TH BGCOLOR=\"BLACK\"><FONT COLOR=\"WHITE\">" +
       bid.getItemName() + "</FONT>\n" +
       "  <TR><TH>Item ID: " +
```

Listing 4.16 BidServlet.java *(continued)*

```
      bid.getItemID() + "\n" +
      "  <TR><TH>Name: " +
      bid.getBidderName() + "\n" +
      "  <TR><TH>Email address: " +
      bid.getEmailAddress() + "\n" +
      "  <TR><TH>Bid price: $" +
      bid.getBidPrice() + "\n" +
      "  <TR><TH>Auto-increment price: " +
      bid.isAutoIncrement() + "\n" +
      "</TABLE></CENTER></BODY></HTML>");
}

/** If the required data is totally missing, show a blank
 *  form. If the required data is partially missing,
 *  warn the user, fill in form fields that already have
 *  values, and prompt user for missing fields.
 */

private void showEntryForm(HttpServletRequest request,
                           HttpServletResponse response,
                           BidInfo bid)
    throws ServletException, IOException {
  boolean isPartlyComplete = bid.isPartlyComplete();
  response.setContentType("text/html");
  PrintWriter out = response.getWriter();
  String title =
    "Welcome to Auctions-R-Us. Please Enter Bid.";
  out.println
    (DOCTYPE +
     "<HTML>\n" +
     "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
     "<BODY BGCOLOR=\"#FDF5E6\"><CENTER>\n" +
     "<H1>" + title + "</H1>\n" +
     warning(isPartlyComplete) +
     "<FORM>\n" +
     inputElement("Item ID", "itemID",
                  bid.getItemID(), isPartlyComplete) +
     inputElement("Item Name", "itemName",
                  bid.getItemName(), isPartlyComplete) +
     inputElement("Your Name", "bidderName",
                  bid.getBidderName(), isPartlyComplete) +
     inputElement("Your Email Address", "emailAddress",
                  bid.getEmailAddress(), isPartlyComplete) +
     inputElement("Amount Bid", "bidPrice",
                  bid.getBidPrice(), isPartlyComplete) +
```

Listing 4.16 BidServlet.java *(continued)*

```
      checkbox("Auto-increment bid to match other bidders?",
               "autoIncrement", bid.isAutoIncrement()) +
      "<INPUT TYPE=\"SUBMIT\" VALUE=\"Submit Bid\">\n" +
      "</CENTER></BODY></HTML>");
}

private void submitBid(BidInfo bid) {
  // Some application-specific code to record the bid.
  // The point is that you pass in a real object with
  // properties populated, not a bunch of strings.
}

private String warning(boolean isFormPartlyComplete) {
  if(isFormPartlyComplete) {
    return("<H2>Required Data Missing! " +
           "Enter and Resubmit.</H2>\n");
  } else {
    return("");
  }
}

/** Create a textfield for input, prefaced by a prompt.
 *  If this particular textfield is missing a value but
 *  other fields have values (i.e., a partially filled form
 *  was submitted), then add a warning telling the user that
 *  this textfield is required.
 */

private String inputElement(String prompt,
                            String name,
                            String value,
                            boolean shouldPrompt) {
  String message = "";
  if (shouldPrompt && ((value == null) || value.equals(""))) {
    message = "<B>Required field!</B> ";
  }
  return(message + prompt + ": " +
         "<INPUT TYPE=\"TEXT\" NAME=\"" + name + "\"" +
         " VALUE=\"" + value + "\"><BR>\n");
}

private String inputElement(String prompt,
                            String name,
                            double value,
                            boolean shouldPrompt) {
```

Listing 4.16 BidServlet.java *(continued)*

```
    String num;
    if (value == 0.0) {
      num = "";
    } else {
      num = String.valueOf(value);
    }
    return(inputElement(prompt, name, num, shouldPrompt));
  }

  private String checkbox(String prompt,
                          String name,
                          boolean isChecked) {
    String result =
      prompt + ": " +
      "<INPUT TYPE=\"CHECKBOX\" NAME=\"" + name + "\"";
    if (isChecked) {
      result = result + " CHECKED";
    }
    result = result + "><BR>\n";
    return(result);
  }

  private final String DOCTYPE =
    "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
    "Transitional//EN\">\n";
}
```

Listing 4.17 BidInfo.java

```
package coreservlets.beans;

import coreservlets.*;

/** Bean that represents information about a bid at
 *  an auction site. Used to demonstrate redisplay of forms
 *  that have incomplete data.
 */

public class BidInfo {
  private String itemID = "";
  private String itemName = "";
  private String bidderName = "";
  private String emailAddress = "";
  private double bidPrice = 0;
  private boolean autoIncrement = false;
```

Listing 4.17 BidInfo.java *(continued)*

```
public String getItemName() {
  return(itemName);
}

public void setItemName(String itemName) {
  this.itemName = ServletUtilities.filter(itemName);
}

public String getItemID() {
  return(itemID);
}

public void setItemID(String itemID) {
  this.itemID = ServletUtilities.filter(itemID);
}

public String getBidderName() {
  return(bidderName);
}

public void setBidderName(String bidderName) {
  this.bidderName =  ServletUtilities.filter(bidderName);
}

public String getEmailAddress() {
  return(emailAddress);
}

public void setEmailAddress(String emailAddress) {
  this.emailAddress = ServletUtilities.filter(emailAddress);
}

public double getBidPrice() {
  return(bidPrice);
}

public void setBidPrice(double bidPrice) {
  this.bidPrice = bidPrice;
}

public boolean isAutoIncrement() {
  return(autoIncrement);
}
```

Listing 4.17 BidInfo.java *(continued)*

```
  public void setAutoIncrement(boolean autoIncrement) {
    this.autoIncrement = autoIncrement;
  }

  /** Has all the required data been entered? Everything except
      autoIncrement must be specified explicitly (autoIncrement
      defaults to false).
  */

  public boolean isComplete() {
    return(hasValue(getItemID()) &&
           hasValue(getItemName()) &&
           hasValue(getBidderName()) &&
           hasValue(getEmailAddress()) &&
           (getBidPrice() > 0));
  }

  /** Has any of the data been entered? */

  public boolean isPartlyComplete() {
    boolean flag =
           (hasValue(getItemID()) ||
           hasValue(getItemName()) ||
           hasValue(getBidderName()) ||
           hasValue(getEmailAddress()) ||
           (getBidPrice() > 0) ||
           isAutoIncrement());
    return(flag);
  }

  private boolean hasValue(String val) {
    return((val != null) && (!val.equals("")));
  }
}
```

Handling the Client Request: HTTP Request Headers

Topics in This Chapter

- Reading HTTP request headers
- Building a table of all the request headers
- Understanding the various request headers
- Reducing download times by compressing pages
- Differentiating among types of browsers
- Customizing pages according to how users got there
- Accessing the standard CGI variables

Chapter 5

One of the keys to creating effective servlets is understanding how to manipulate the HyperText Transfer Protocol (HTTP). Thoroughly understanding this protocol is not an esoteric, theoretical concept, but rather a practical issue that can have an 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 the most important HTTP 1.1 request headers, summarizing how and why they would be used in a servlet. As we see later, request headers are read and applied the same way in JSP pages as they are in servlets.

Note that HTTP request headers are distinct from the form (query) data discussed in the previous chapter. 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/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
```

5.1 Reading Request Headers

Reading headers is straightforward; just call the `getHeader` method of `HttpServletRequest` with the name of the header. This call returns a `String` if the specified header was supplied in the current request, `null` otherwise. In HTTP 1.0, all request headers are optional; in HTTP 1.1, only `Host` is required. So, always check for `null` before using a request header.

Core Approach

Always check that the result of `request.getHeader` is non-`null` before using it.

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 Chapter 8 (Handling Cookies).
- **`getAuthType` and `getRemoteUser`**
  The `getAuthType` and `getRemoteUser` methods break the `Authorization` header into its component pieces.
- **`getContentLength`**
  The `getContentLength` method returns the value of the `Content-Length` header (as an `int`).
- **`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 Section 5.2 (Making a Table of All Request Headers).
- **`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 (i.e., the first line in the example request just shown), also by means of methods in `HttpServletRequest`. Here is a summary of the four 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?subject=jsp, `getRequestURI` would return `"/servlet/search.BookSearch"`.
- **`getQueryString`**
  The `getQueryString` method returns the form data. For example, with http://randomhost.com/servlet/search.BookSearch?subject=jsp, `getQueryString` would return `"subject=jsp"`.
- **`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 (Chapter 7) that are specific to HTTP 1.1.

5.2 Making a Table of All Request Headers

Listing 5.1 shows a servlet that simply creates a table of all the headers it receives, along with their associated values. It accomplishes this task by calling `request.getHeaderNames` to obtain an `Enumeration` of headers in the current request. It then loops down the `Enumeration`, puts the header name in the left table cell, and puts the result of `getHeader` in the right table cell. Recall that `Enumeration` is a standard interface in Java; it is in the `java.util` package and contains just two methods: `hasMoreElements` and `nextElement`.

The servlet also prints three components of the main request line (method, URI, and protocol). Figures 5–1 and 5–2 show typical results with Netscape and Internet Explorer.

Listing 5.1 ShowRequestHeaders.java

```
package coreservlets;

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

/** Shows all the request headers sent on the current 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";
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<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>" +
```

Listing 5.1 ShowRequestHeaders.java *(continued)*

```
                request.getProtocol() + "<BR><BR>\n" +
                "<TABLE BORDER=1 ALIGN=\"CENTER\">\n" +
                "<TR BGCOLOR=\"#FFAD00\">\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</BODY></HTML>");
  }

  /** Since this servlet is for debugging, have it
   *  handle GET and POST identically.
   */

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

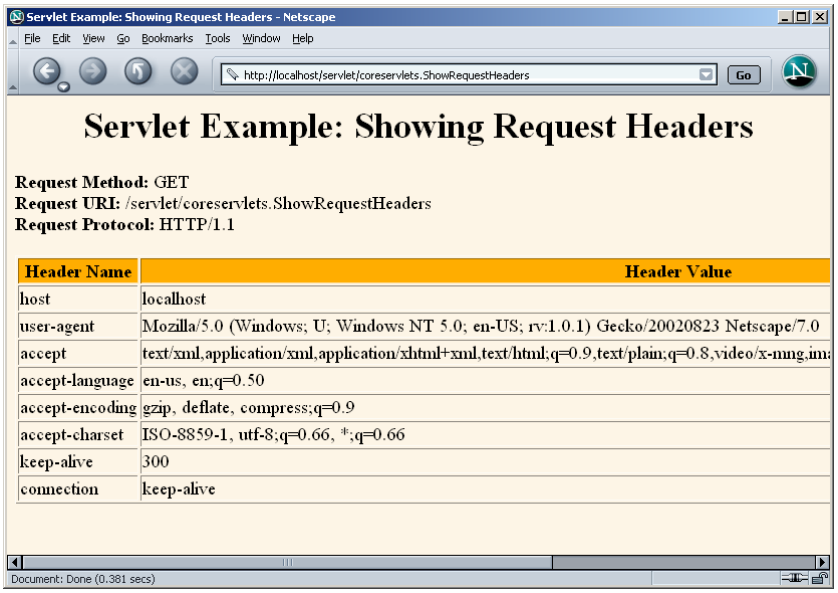


Figure 5–1 Request headers sent by Netscape 7 on Windows 2000.

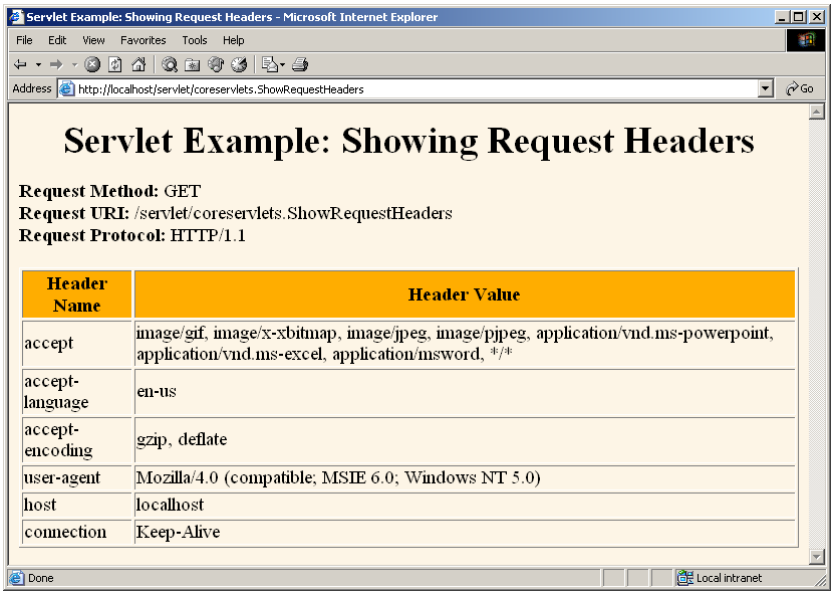


Figure 5–2 Request headers sent by Internet Explorer 6 on Windows 2000.

5.3 Understanding 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 section summarizes the headers most often used by servlets; 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. Note that HTTP 1.1 supports a superset of the headers permitted in HTTP 1.0.

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 not all browsers support PNG. If you have images in both formats, your servlet can call `request.getHeader("Accept")`, check for `image/png`, and if it finds a match, use *blah*.png filenames in all the `IMG` elements it generates. Otherwise, it would just use *blah*.gif.

See Table 7.1 in Section 7.2 (Understanding HTTP 1.1 Response Headers) for the names and meanings of the common MIME types.

Note that Internet Explorer 5 and 6 have a bug whereby the `Accept` header is sent improperly when you reload a page. It is sent properly in 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 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 cost of decoding is likely to be small compared with the savings in transmission time. See Section 5.4 in which gzip compression is used to reduce download times by a factor of more than 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 the chapters on Web application security in Volume 2 of this book.

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, thus 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 to cooperate with the server to handle persistent connections. Consequently, the servlet's job is just to make it *possible* for the server to use persistent connections; the servlet does so by setting the `Content-Length` response header. For details, see Chapter 7 (Generating the Server Response: HTTP Response Headers).

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 Chapter 4), you rarely use this header explicitly.

Cookie

This header returns cookies to servers that previously sent them to the browser. Never read this header directly because doing so would require cumbersome low-level parsing; use `request.getCookies` instead. For details, see Chapter 8 (Handling 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 and 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. Because of the widespread use of virtual hosting (one computer handling Web sites for multiple domain names), 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.

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. For an example, see the lottery numbers servlet in Section 3.6 (The Servlet Life Cycle).

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. Most 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, for identifying when users first enter your application, or simply for keeping track of where your traffic comes from. In the last case, most people 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. Also, note that, owing to a spelling mistake by one of the original HTTP authors, this header is `Referer`, not the expected `Referrer`.

Finally, note that some browsers (Opera), ad filters (Web Washer), and personal firewalls (Norton) screen out this header. Besides, even in normal situations, the header is only set when the user follows a link. So, be sure to follow the approach you should be using with all headers anyhow: check for `null` before using the header.

See Section 5.6 (Changing the Page According to How the User Got There) for details and an example.

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 use 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.

However, the `User-Agent` header is quite useful for distinguishing among different *categories* of client. For example, Japanese developers might see whether the `User-Agent` is an Imode cell phone (in which case they would redirect to a chtml page), a Skynet cell phone (in which case they would redirect to a wml page), or a Web browser (in which case they 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. The Opera browser does the same thing. This deliberate misidentification is done for compatibility with JavaScript; JavaScript developers often use the `User-Agent` header to determine which JavaScript features are supported. So, if you want to differentiate Netscape from Internet Explorer, you have to check for the string "MSIE" or something more specific, not just the string "Mozilla." 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.

See Section 5.5 (Differentiating Among Different Browser Types) for details and an example.

5.4 Sending Compressed Web Pages

Gzip is a text compression scheme that can dramatically reduce the size of HTML (or plain text) pages. Most recent browsers know how to handle gzipped content, so the server can compress the document and send the smaller document over the network, after which the browser will automatically reverse the compression (no user action required) and treat the result in the normal manner. Sending such compressed content can be a real time saver since the time required to compress the document on the server and then uncompress it on the client is typically dwarfed by the time saved in download time, especially when dialup connections are used.

DILBERT reprinted by permission of United Feature Syndicate, Inc.

However, although most recent browsers support this capability, not all do. If you send gzipped content to browsers that don't support this capability, the browsers will not be able to display the page at all. Fortunately, browsers that support this feature indicate that they do so by setting the `Accept-Encoding` request header. Browsers that support content encoding include most versions of Netscape for Unix, most versions of Internet Explorer for Windows, and Netscape 4.7 and later for Windows. Earlier Netscape versions on Windows and Internet Explorer on non-Windows platforms generally do not support content encoding.

Listing 5.2 shows a servlet that checks the `Accept-Encoding` header, sending a compressed Web page to clients that support gzip encoding (as determined by the `isGzipSupported` method of Listing 5.3) and sending a regular Web page to those that don't. The result (see Figure 5–3) yielded a compression of over *300*-fold and a speedup of more than a factor of *10* when a dialup connection was used. In repeated tests with Netscape and Internet Explorer on a 28.8K modem connection, the compressed page averaged less than 5 seconds to completely download, whereas the uncompressed page consistently took more than 50 seconds. Results were less dramatic with faster connections, but the improvement was still significant. Gzip compression is such a useful technique that we later present a filter that lets you apply gzip compression to designated servlets or JSP pages without changing the actual code of the individual resources. For details, see the chapter on servlet and JSP filters in Volume 2 of this book.

Core Tip

Gzip compression can dramatically reduce the download time of long text pages.

Implementing compression is straightforward since support for the gzip format is built in to the Java programming language by classes in `java.util.zip`. The servlet first checks the `Accept-Encoding` header to see if it contains an entry for gzip. If so, it uses a `PrintWriter` wrapped around a `GZIPOutputStream` and specifies `gzip` as the value of the `Content-Encoding` response header. If gzip is not supported, the servlet uses the normal `PrintWriter` and omits the `Content-Encoding` header. To make it easy to compare regular and compressed performance with the same browser, we also added a feature whereby we can suppress compression by including `?disableGzip` at the end of the URL.

Listing 5.2 LongServlet.java

```
package coreservlets;

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

/** Servlet with <B>long</B> output. Used to test
 *  the effect of the gzip compression.
 */

public class LongServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");

    // Change the definition of "out" depending on whether
    // or not gzip is supported.
    PrintWriter out;
    if (GzipUtilities.isGzipSupported(request) &&
        !GzipUtilities.isGzipDisabled(request)) {
      out = GzipUtilities.getGzipWriter(response);
      response.setHeader("Content-Encoding", "gzip");
    } else {
      out = response.getWriter();
    }

    // Once "out" has been assigned appropriately, the
    // rest of the page has no dependencies on the type
    // of writer being used.
```

Listing 5.2 LongServlet.java *(continued)*

```
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    String title = "Long Page";
    out.println
      (docType +
       "<HTML>\n" +
       "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
       "<BODY BGCOLOR=\"#FDF5E6\">\n" +
       "<H1 ALIGN=\"CENTER\">" + title + "</H1>\n");
    String line = "Blah, blah, blah, blah, blah. " +
                  "Yadda, yadda, yadda, yadda.";
    for(int i=0; i<10000; i++) {
      out.println(line);
    }
    out.println("</BODY></HTML>");
    out.close(); // Needed for gzip; optional otherwise.
  }
}
```

Listing 5.3 GzipUtilities.java

```
package coreservlets;

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

/** Three small static utilities to assist with gzip encoding.
 *  <UL>
 *    <LI>isGzipSupported: does the browser support gzip?
 *    <LI>isGzipDisabled: has the user passed in a flag
 *        saying that gzip encoding should be disabled for
 *        this request? (Useful so that you can measure
 *        results with and without gzip on the same browser).
 *    <LI>getGzipWriter: return a gzipping PrintWriter.
 *  </UL>
 */

public class GzipUtilities {

  /** Does the client support gzip? */
```

Listing 5.3 GzipUtilities.java *(continued)*

```
  public static boolean isGzipSupported
      (HttpServletRequest request) {
    String encodings = request.getHeader("Accept-Encoding");
    return((encodings != null) &&
           (encodings.indexOf("gzip") != -1));
  }

  /** Has user disabled gzip (e.g., for benchmarking)? */

  public static boolean isGzipDisabled
      (HttpServletRequest request) {
    String flag = request.getParameter("disableGzip");
    return((flag != null) && (!flag.equalsIgnoreCase("false")));
  }

  /** Return gzipping PrintWriter for response. */

  public static PrintWriter getGzipWriter
      (HttpServletResponse response) throws IOException {
    return(new PrintWriter
             (new GZIPOutputStream
               (response.getOutputStream())));
  }
}
```

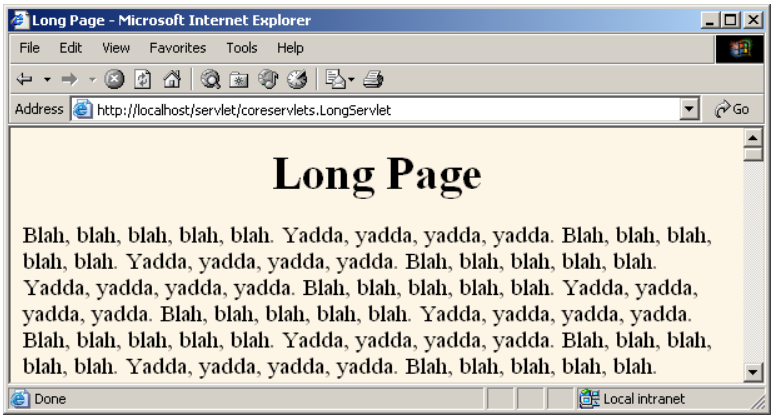


Figure 5–3 Since the Windows version of Internet Explorer 6 supports gzip, this page was sent gzipped over the network and automatically reconstituted by the browser, resulting in a large saving in download time.

5.5 Differentiating Among Different Browser Types

The `User-Agent` header identifies the specific browser that is making the request. Although use of this header appears straightforward at first glance, a few subtleties are involved:

- **Use `User-Agent` only when necessary.** Otherwise, you will have difficult-to-maintain code that consists of tables of browser versions and associated capabilities. For example, instead of remembering that the Windows version of Internet Explorer 5 supports gzip compression but the MacOS version doesn't, check the `Accept-Encoding` header. Instead of remembering which browsers support Java and which don't, use the `APPLET` tag with fallback code between `<APPLET>` and `</APPLET>`.
- **Check for `null`.** Sure, all major browser versions send the `User-Agent` header. But, the header is not *required* by the HTTP 1.1 specification, some browsers let you disable it (e.g., Opera), and custom clients (e.g., Web spiders or link verifiers) might not use the header at all. In fact, you should *always* check that the result of `request.getHeader` is non-`null` before trying to use it, regardless of which header you are dealing with.
- **To differentiate between Netscape and Internet Explorer, check for "MSIE," not "Mozilla."** Both Netscape and Internet Explorer say "Mozilla" at the beginning of the header, even though Mozilla is the Godzilla-like Netscape mascot. This characteristic is for compatibility with JavaScript.
- **Note that the header can be faked.** Some browsers let the user change the value of this header. Even if the browser didn't allow this, the user could always use a custom client. If a client fakes this header, the servlet cannot tell the difference.

Listing 5.4 shows a servlet that sends browser-specific insults to users. For the sake of simplicity, it assumes that Internet Explorer and Netscape are the only two browsers being used. Specifically, it assumes that any browser whose `User-Agent` contains "MSIE" is Internet Explorer and any whose `User-Agent` does not is Netscape. Figures 5–4 and 5–5 show the results.

Listing 5.4 BrowserInsult.java

```
package coreservlets;

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

/** Servlet that gives browser-specific insult.
 *  Illustrates how to use the User-Agent
 *  header to tell browsers apart.
 */

public class BrowserInsult extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title, message;
    // Assume for simplicity that Netscape and IE are
    // the only two browsers.
    String userAgent = request.getHeader("User-Agent");
    if ((userAgent != null) &&
        (userAgent.indexOf("MSIE") != -1)) {
      title = "Microsoft Minion";
      message = "Welcome, O spineless slave to the " +
                "mighty empire.";
    } else {
      title = "Hopeless Netscape Rebel";
      message = "Enjoy it while you can. " +
                "You <I>will</I> be assimilated!";
    }
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                message + "\n" +
                "</BODY></HTML>");
  }
}
```

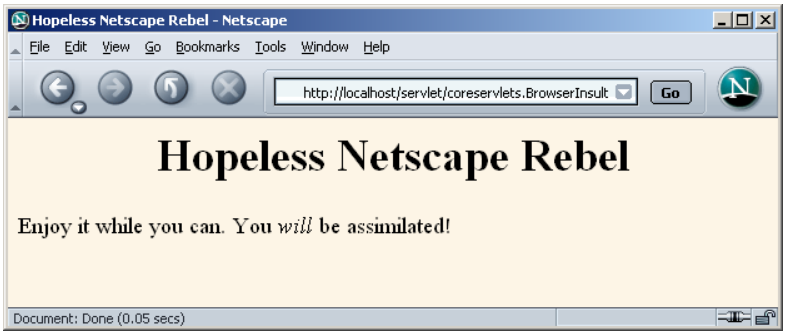


Figure 5–4 The `BrowserInsult` servlet as viewed by a Netscape user.

Figure 5–5 The `BrowserInsult` servlet as viewed by an Internet Explorer user.

5.6 Changing the Page According to How the User Got There

The `Referer` header designates the location of the page users were on when they clicked a link to get to the current page. If users simply type the address of a page, the browser sends no `Referer` at all and `request.getHeader("Referer")` returns `null`.

This header enables you to customize the page depending on how the user reached it. For example, you could use this header to do the following:

- Create a jobs/careers site that takes on the look and feel of the associated site that links to it.
- Change the content of a page depending on whether the link came from inside or outside the firewall. (Do not use this trick for secure applications, however; the `Referer` header, like all headers, is easily forged.)

- Supply links that take users back to the page they came from.
- Track the effectiveness of banner ads or record click-through rates from various different sites that display your ads.

Listing 5.5 shows a servlet that uses the `Referer` header to customize the image it displays. If the address of the referring page contains the string "JRun," the servlet displays the logo of Macromedia JRun. If the address contains the string "Resin," the servlet displays the logo of Caucho Resin. Otherwise, the servlet displays the logo of Apache Tomcat. The servlet also displays the address of the referring page.

Listing 5.6 shows the HTML pages used to link to the servlet. We created three *identical* pages named JRun-Referer.html, Resin-Referer.html, and Tomcat-Referer.html; the servlet uses the name of the referring page, not form data, to distinguish among the three. Recall that HTML pages are placed in the top-level directory of your Web application (or an arbitrary subdirectory thereof), whereas servlet code is placed in a subdirectory of WEB-INF/classes that matches the package name. So, for example, with Tomcat and the default Web application, the HTML pages are placed in *install_dir*/webapps/ROOT/request-headers/ and accessed with URLs of the form http://*hostname*/request-headers/*Xxx*-Referer.html.

Figures 5–6 through 5–9 show some representative results.

Listing 5.5 CustomizeImage.java

```
package coreservlets;

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

/** Servlet that displays referer-specific image. */

public class CustomizeImage extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String referer = request.getHeader("Referer");
    if (referer == null) {
      referer = "<I>none</I>";
    }
    String title = "Referring page: " + referer;
    String imageName;
    if (contains(referer, "JRun")) {
      imageName = "jrun-powered.gif";
```

Listing 5.5 CustomizeImage.java *(continued)*

```
    } else if (contains(referer, "Resin")) {
      imageName = "resin-powered.gif";
    } else {
      imageName = "tomcat-powered.gif";
    }
    String imagePath = "../request-headers/images/" + imageName;
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<CENTER><H2>" + title + "</H2>\n" +
                "<IMG SRC=\"" + imagePath + "\">\n" +
                "</CENTER></BODY></HTML>");
  }

  private boolean contains(String mainString,
                           String subString) {
    return(mainString.indexOf(subString) != -1);
  }
}
```

Listing 5.6 JRun-Referer.html (identical to Tomcat-Referer.html and Resin-Referer.html)

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD><TITLE>Referer Test</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN="CENTER">Referer Test</H1>
Click <A HREF="/servlet/coreservlets.CustomizeImage">here</A>
to visit the servlet.
</BODY></HTML>
```

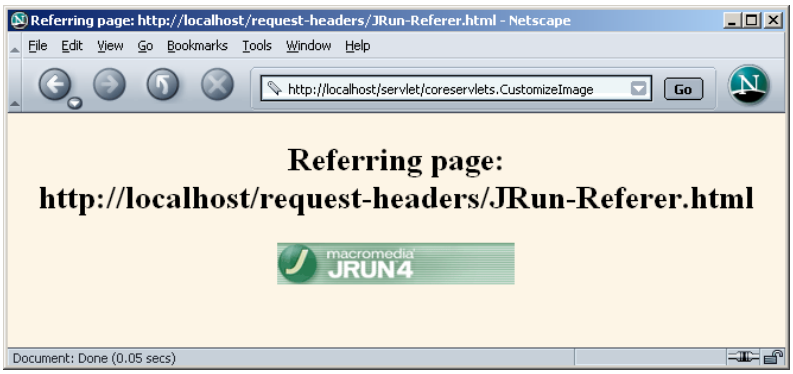


Figure 5–6 The `CustomizeImage` servlet when the address of the referring page contains the string "JRun."

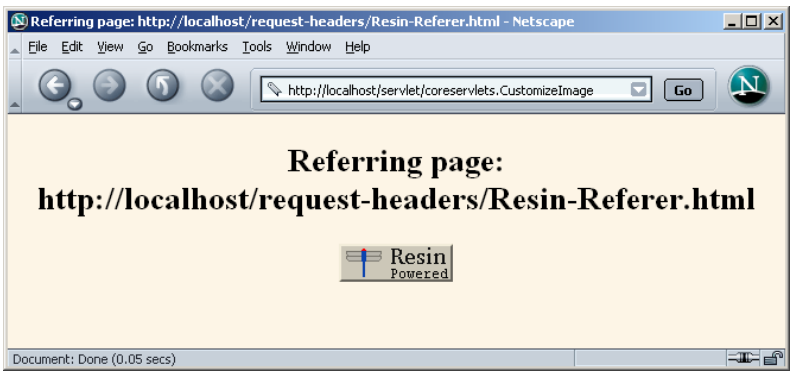


Figure 5–7 The `CustomizeImage` servlet when the address of the referring page contains the string "Resin."

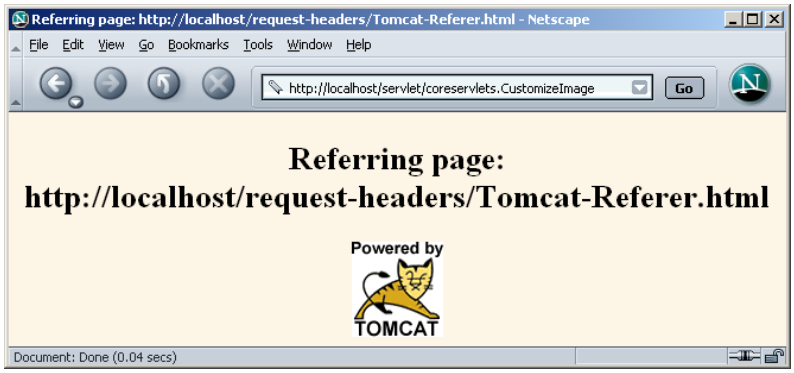


Figure 5–8 The `CustomizeImage` servlet when the address of the referring page contains neither "JRun" nor "Resin."

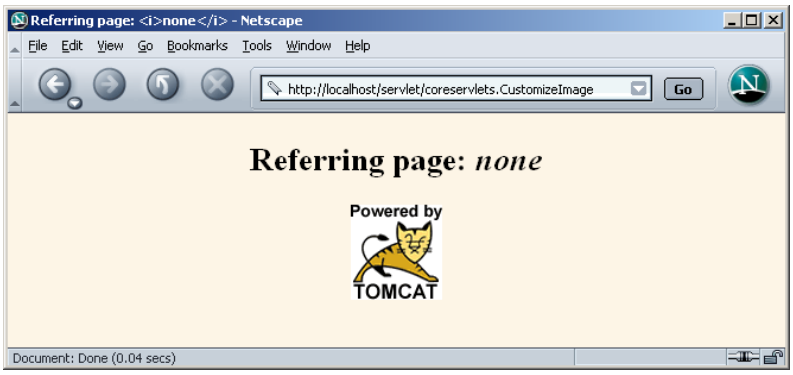


Figure 5–9 The `CustomizeImage` servlet when the `Referer` header is missing. When using the `Referer` header, always handle the case in which the result of `getHeader` is `null`.

5.7 Accessing 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 (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.

Servlet Equivalent of CGI Variables

For each standard CGI variable, this subsection summarizes its purpose and the means of accessing it from a servlet. Assume `request` is the `HttpServletRequest` supplied to the `doGet` and `doPost` methods.

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.getContentLength())` or `request.getHeader("Content-Length")`. You'll probably want to just call `request.getContentLength()`, which returns an `int`.

CONTENT_TYPE

`CONTENT_TYPE` designates the MIME type of attached data, if specified. See Table 7.1 in Section 7.2 (Understanding HTTP 1.1 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("/")`. 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` are how CGI programs access arbitrary HTTP request headers. The `Cookie` header becomes `HTTP_COOKIE`, `User-Agent` becomes `HTTP_USER_AGENT`, `Referer` becomes `HTTP_REFERER`, and so forth. Servlets should just use `request.getHeader` or one of the shortcut methods described in Section 5.1 (Reading 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/coreservlets.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()`.

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 5.1 (Reading 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., `white-house.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()`. For decoding `Authorization` information directly in servlets, see the chapters on Web application security in Volume 2 of this book.

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()`.

SCRIPT_NAME

This variable specifies the path to the servlet, 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 by means of `getServletContext().getServerInfo()`.

A Servlet That Shows the CGI Variables

Listing 5.7 presents a servlet that creates a table showing the values of all the CGI variables other than `HTTP_XXX_YYY`, which are just the HTTP request headers described in Section 5.3. Figure 5–10 shows the result for a typical request.

Listing 5.7 ShowCGIVariables.java

```
package coreservlets;

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

/** Creates a table showing the current value of each
 *  of the standard CGI variables.
 */

public class ShowCGIVariables extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String[][] variables =
      { { "AUTH_TYPE", request.getAuthType() },
        { "CONTENT_LENGTH",
          String.valueOf(request.getContentLength()) },
        { "CONTENT_TYPE", request.getContentType() },
        { "DOCUMENT_ROOT",
          getServletContext().getRealPath("/") },
        { "PATH_INFO", request.getPathInfo() },
        { "PATH_TRANSLATED", request.getPathTranslated() },
        { "QUERY_STRING", request.getQueryString() },
        { "REMOTE_ADDR", request.getRemoteAddr() },
        { "REMOTE_HOST", request.getRemoteHost() },
        { "REMOTE_USER", request.getRemoteUser() },
        { "REQUEST_METHOD", request.getMethod() },
        { "SCRIPT_NAME", request.getServletPath() },
        { "SERVER_NAME", request.getServerName() },
        { "SERVER_PORT",
          String.valueOf(request.getServerPort()) },
        { "SERVER_PROTOCOL", request.getProtocol() },
        { "SERVER_SOFTWARE",
          getServletContext().getServerInfo() }
      };
    String title = "Servlet Example: Showing CGI Variables";
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
```

Listing 5.7 ShowCGIVariables.java *(continued)*

```
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<CENTER>\n" +
                "<H1>" + title + "</H1>\n" +
                "<TABLE BORDER=1>\n" +
                "  <TR BGCOLOR=\"#FFAD00\">\n" +
                "    <TH>CGI Variable Name<TH>Value");
    for(int i=0; i<variables.length; i++) {
      String varName = variables[i][0];
      String varValue = variables[i][1];
      if (varValue == null)
        varValue = "<I>Not specified</I>";
      out.println("  <TR><TD>" + varName + "<TD>" + varValue);
    }
    out.println("</TABLE></CENTER></BODY></HTML>");
  }

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

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

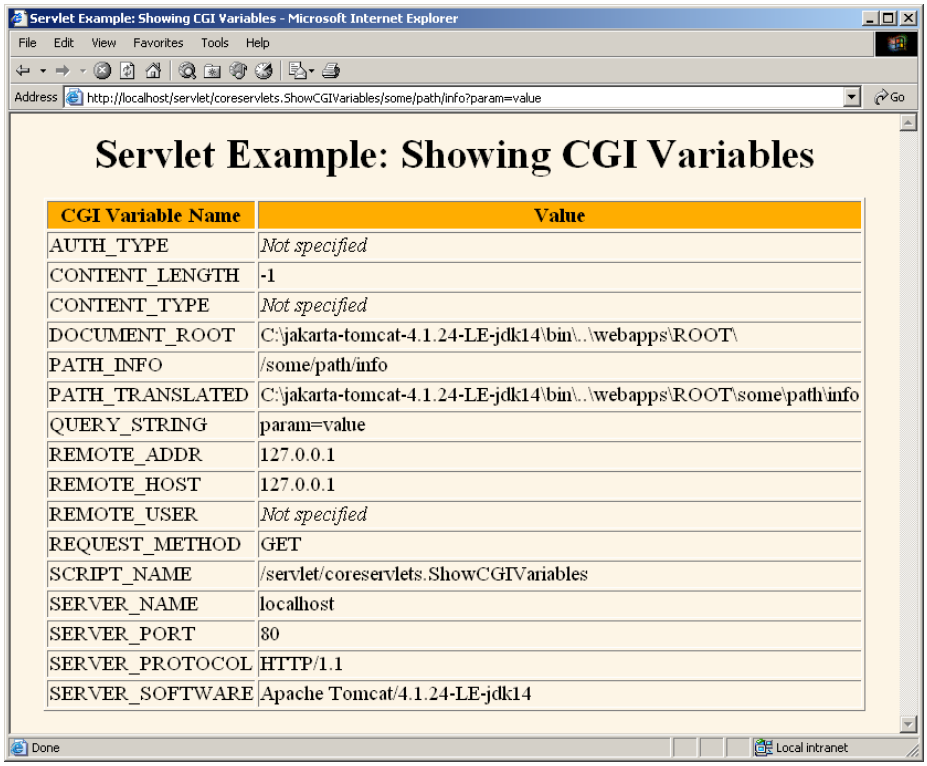


Figure 5–10 The standard CGI variables for a typical request.

Generating the Server Response: HTTP Status Codes

Topics in This Chapter

- Format of the HTTP response
- How to set status codes
- What the status codes are good for
- Shortcut methods for redirection and error pages
- A servlet that redirects users to browser-specific pages
- A front end to various search engines

Chapter 6

As we saw in the previous chapter, a request from a browser or other client consists of an HTTP command (usually `GET` or `POST`), zero or more request headers (one or more in HTTP 1.1, since `Host` is required), a blank line, and, only in the case of `POST` requests, some query data. A typical request looks like the following.

```
GET /servlet/SomeName HTTP/1.1
Host: ...
Header2: ...
...
HeaderN:
  (Blank Line)
```

When a Web server responds to a request, the response typically consists of a status line, some response headers, a blank line, and the document. A typical response looks like this:

```
HTTP/1.1 200 OK
Content-Type: text/html
Header2: ...
...
HeaderN: ...
  (Blank Line)
<!DOCTYPE ...>
<HTML>
<HEAD>...</HEAD>
<BODY>
...
</BODY></HTML>
```

The status line consists of the HTTP version (`HTTP/1.1` in the preceding example), 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, 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 various status codes essentially indicate failure or redirection (and thus 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 chapter summarizes the most important status codes and describes what can be accomplished with them; the following chapter discusses the response headers.

6.1 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* want to, they use `response.setStatus`, `response.sendRedirect`, or `response.sendError`.

Setting Arbitrary Status Codes: setStatus

When you want to set an arbitrary status code, 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`. The reason is that an HTTP response consists of the status line, one or more headers, a blank line, and the actual document, *in that order*. Servlets do not necessarily buffer the document, so you have to either set the status code before 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 readability and to avoid typos, 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 is one minor exception, however: the constant for code 302 is derived from the message defined by HTTP 1.0 (Moved Temporarily), not the HTTP 1.1 message (Found).

Setting 302 and 404 Status Codes: sendRedirect and sendError

Although the general method of setting status codes is simply to call `response.setStatus(int)`, there are two common cases for which a shortcut method in `HttpServletResponse` is provided. Just be aware that both of these methods throw `IOException`, whereas `setStatus` does not. Since the `doGet` and `doPost` methods already throw `IOException`, this difference only matters if you pass the response object to another method.

- **`public void sendRedirect(String url)`**
  The 302 status code directs the browser to connect to a new location. The `sendRedirect` method generates a 302 response along with a `Location` header giving the URL of the new document. 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.
- **`public void sendError(int code, String message)`**
  The 404 status code is used when no document is found on the server. 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.

Setting a status code does not necessarily mean that you omit the 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*.

6.2 HTTP 1.1 Status Codes

In this section we 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 capabilities 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.

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 Consortium, 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.
- **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 (`SC_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.

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. Thus, this status code is used to instruct 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. Note that the browser reconnects to the new URL immediately; no intermediate output is displayed. This behavior distinguishes *redirects* from *refreshes* where an intermediate page is temporarily displayed (see the next chapter for details on the `Refresh` header). With redirects, another site, not the servlet itself, generates the results. So why use a servlet at all? Redirects are useful for the following tasks:

- **Computing destinations.** If you know the final destination for the user in advance, your hypertext link or HTML form could send the user directly there. But, if you need to look at the data before deciding where to obtain the necessary results, a redirection is useful. For example, you might want to send users to a standard site that gives information on stocks, but you need to look at the stock symbol before deciding whether to send them to the New York Stock Exchange, NASDAQ, or a non-U.S. site.
- **Tracking user behavior.** If you send users a page that contains a hypertext link to another site, you have no way to know if they actually click on the link. But perhaps this information is important in analyzing the usefulness of the different links you send them. So, instead of sending users the direct link, you can send them a link to your own site, where you can then record some information and then redirect them to the real site. For example, several search engines use this trick to determine which of the results they display are most popular.
- **Performing side effects.** What if you want to send users to a certain site but set a cookie on the user's browser first? No problem: return both a `Set-Cookie` response header (by means of `response.addCookie`—see Chapter 8) *and* a 302 status code (by means of `response.sendRedirect`).

The 302 status code 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, `sendRedirect` can handle relative URLs, automatically translating them into absolute ones.

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

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`. See status code 307. 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 signify 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 the `LotteryNumbers` servlet in Section 3.6 (The Servlet Life Cycle).

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. 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 but that the request did not have proper identifying

information in the `Authorization` header. The response must include a `WWW-Authenticate` header. For details, see the chapter on programmatic Web application security in Volume 2 of this book.

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.

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 (to see them, just make up a nonexistent URL at either site). 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/. We find http://www.plinko.net/404/links.asp?type=cat&key=13 (amusing 404 error messages) particularly funny.

Unfortunately, however, the default behavior of Internet Explorer in version 5 and later is to ignore the error page you send back and to display its own static (and relatively useless) error message, even though doing so explicitly contradicts the HTTP specification. To turn off this setting, go to the Tools menu, select Internet Options, choose the Advanced tab, and make sure the "Show friendly HTTP error messages" box is *not* checked. Regrettably, few users are aware of this setting, so this "feature" prevents most users of Internet Explorer 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 versions 5 and later improperly ignore server-generated error pages.

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 an entire Web application; see Section 2.11 (Web Applications: A Preview) for details.

405 (Method Not Allowed)

A 405 (`SC_METHOD_NOT_ALLOWED`) value signifies 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.

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.

6.3 A Servlet That Redirects Users to Browser-Specific Pages

Recall from Chapter 5 that the `User-Agent` request header designates the specific browser (or cell phone or other client) making the request. Recall further that most major browsers contain the string `Mozilla` in their `User-Agent` header, but only Microsoft Internet Explorer contains the string `MSIE`.

Listing 6.1 shows a servlet that makes use of this fact to send Internet Explorer users to the Netscape home page, and all other users to the Microsoft home page. The servlet accomplishes this task by using the `sendRedirect` method to send a 302 status code and a `Location` response header to the browser. Figures 6–1 and 6–2 show results for Internet Explorer and Netscape, respectively.

Listing 6.1 WrongDestination.java

```
package coreservlets;

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

/** Servlet that sends IE users to the Netscape home page and
 *  Netscape (and all other) users to the Microsoft home page.
 */

public class WrongDestination extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    String userAgent = request.getHeader("User-Agent");
    if ((userAgent != null) &&
        (userAgent.indexOf("MSIE") != -1)) {
      response.sendRedirect("http://home.netscape.com");
    } else {
      response.sendRedirect("http://www.microsoft.com");
    }
  }
}
```

Figure 6–1 Result of http://*host*/servlet/coreservlets.WrongDestination in Internet Explorer.

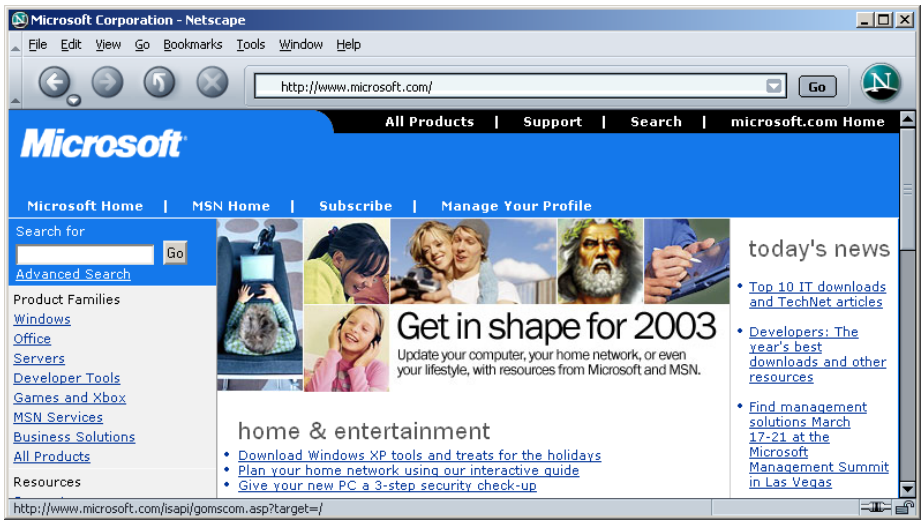


Figure 6–2 Result of http://*host*/servlet/coreservlets.WrongDestination in Netscape.

6.4 A Front End to Various Search Engines

Suppose that you want to make a "one-stop searching" site that lets users search any of the most popular search engines without having to remember many different URLs. You want to let users enter a query, select the search engine, and then send them to that search engine's results page for that query. If users omit the search keywords or fail to select a search engine, you have no site to redirect them to, so you want to display an error page informing them of this fact.

Listing 6.2 (`SearchEngines.java`) presents a servlet that accomplishes these tasks by making use of the 302 (`Found`) and 404 (`Not Found`) status codes—the two most common status codes other than 200. The 302 code is set by the shorthand `sendRedirect` method of `HttpServletResponse`, and 404 is specified by `sendError`.

In this application, a servlet builds an HTML form (see Figure 6–3 and the source code in Listing 6.5) that displays a page to let the user specify a search string and select the search engine to use. When the form is submitted, the servlet extracts those two parameters, constructs a URL with the parameters embedded in a way appropriate to the search engine selected (see the `SearchSpec` and `SearchUtilities` classes of Listings 6.3 and 6.4), and redirects the user to that URL (see Figure 6–4). If the user fails to choose a search engine or specify search terms, an error page informs the client of this fact (see Figure 6–5, but recall warnings about Internet Explorer under the 404 status code in the previous section).

Listing 6.2 SearchEngines.java

```
package coreservlets;

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

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

Listing 6.2 SearchEngines.java *(continued)*

```
public class SearchEngines extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    String searchString = request.getParameter("searchString");
    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, and the
    // getParameter method decodes automatically. But since
    // we're just passing this on to another server, we need to
    // re-encode it to avoid characters that are illegal in
    // URLs. Also note that JDK 1.4 introduced a two-argument
    // version of URLEncoder.encode and deprecated the one-arg
    // version. However, since version 2.3 of the servlet spec
    // mandates only the Java 2 Platform (JDK 1.2 or later),
    // we stick with the one-arg version for portability.
    searchString = URLEncoder.encode(searchString);

    String searchEngineName =
      request.getParameter("searchEngine");
    if ((searchEngineName == null) ||
        (searchEngineName.length() == 0)) {
      reportProblem(response, "Missing search engine name");
      return;
    }
    String searchURL =
      SearchUtilities.makeURL(searchEngineName, searchString);
    if (searchURL != null) {
      response.sendRedirect(searchURL);
    } else {
      reportProblem(response, "Unrecognized search engine");
    }
  }

  private void reportProblem(HttpServletResponse response,
                             String message)
      throws IOException {
    response.sendError(response.SC_NOT_FOUND, message);
  }
}
```

Listing 6.3 SearchSpec.java

```
package coreservlets;

/** Small class that encapsulates how to construct a
 *  search string for a particular search engine.
 */

public class SearchSpec {
  private String name, baseURL;

  public SearchSpec(String name,
                    String baseURL) {
    this.name = name;
    this.baseURL = baseURL;
  }

  /** Builds a URL for the results page by simply concatenating
   *  the base URL (http://...?someVar=") with the URL-encoded
   *  search string (jsp+training).
   */

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

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

Listing 6.4 SearchUtilities.java

```
package coreservlets;

/** Utility with static method to build a URL for any
 *  of the most popular search engines.
 */

public class SearchUtilities {
  private static SearchSpec[] commonSpecs =
    { new SearchSpec("Google",
                     "http://www.google.com/search?q="),
      new SearchSpec("AllTheWeb",
                     "http://www.alltheweb.com/search?q="),
```

Listing 6.4 SearchUtilities.java *(continued)*

```
      new SearchSpec("Yahoo",
                     "http://search.yahoo.com/bin/search?p="),
      new SearchSpec("AltaVista",
                     "http://www.altavista.com/web/results?q="),
      new SearchSpec("Lycos",
                     "search.lycos.com/default.asp?query="),
      new SearchSpec("HotBot",
                     "http://hotbot.com/default.asp?query="),
      new SearchSpec("MSN",
                     "http://search.msn.com/results.asp?q="),
    };

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

  /** Given a search engine name and a search string, builds
   *  a URL for the results page of that search engine
   *  for that query. Returns null if the search engine name
   *  is not one of the ones it knows about.
   */

  public static String makeURL(String searchEngineName,
                               String searchString) {
    SearchSpec[] searchSpecs = getCommonSpecs();
    String searchURL = null;
    for(int i=0; i<searchSpecs.length; i++) {
      SearchSpec spec = searchSpecs[i];
      if (spec.getName().equalsIgnoreCase(searchEngineName)) {
        searchURL = spec.makeURL(searchString);
        break;
      }
    }
    return(searchURL);
  }
}
```

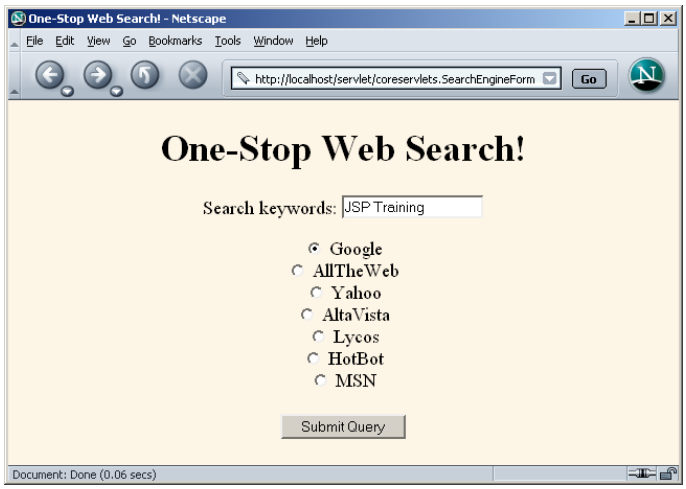


Figure 6–3 Front end to the `SearchEngines` servlet. See Listing 6.5 for the source code.

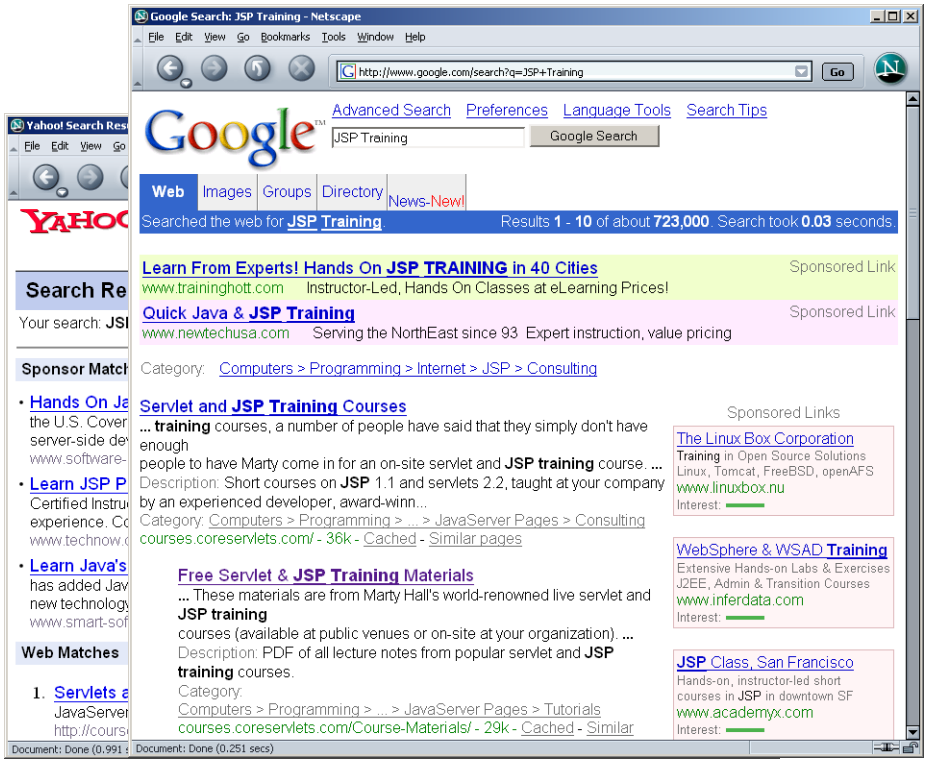


Figure 6–4 Results of the `SearchEngines` servlet when the form of Figure 6–3 is submitted. Although the form is submitted to the `SearchEngines` servlet, that servlet generates no output and the end user sees only the result of the redirection.

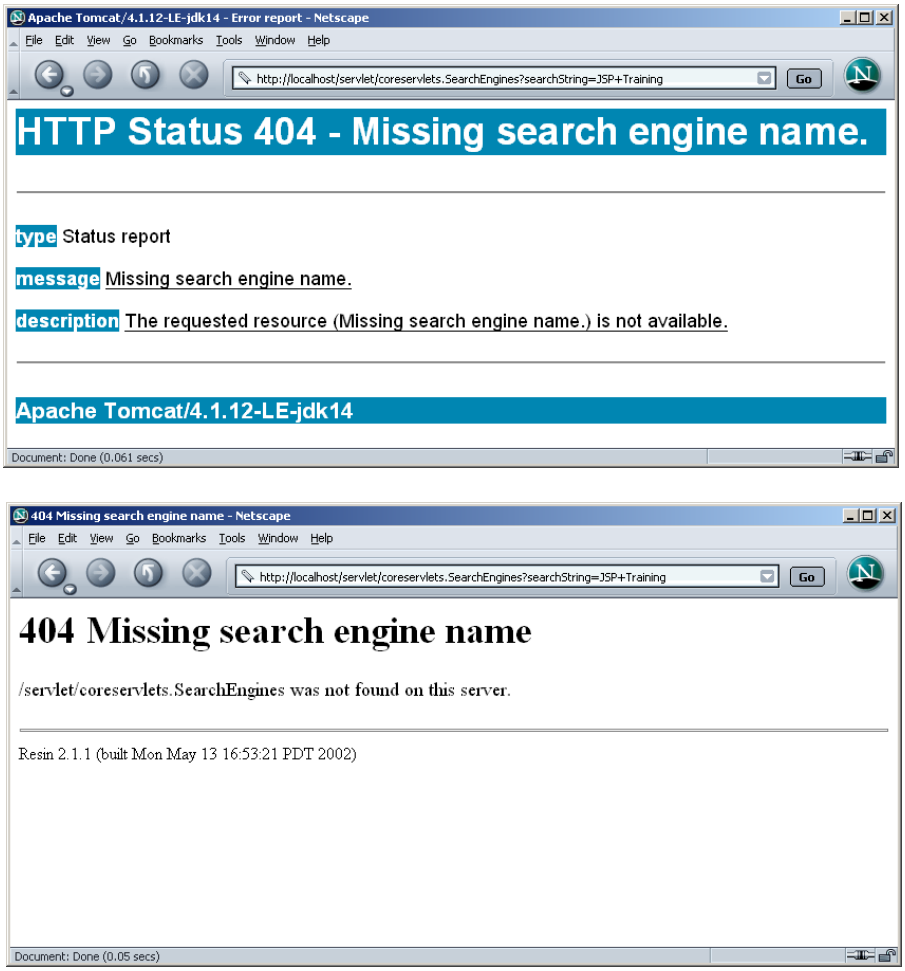


Figure 6–5 Results of the `SearchEngines` servlet upon submission of a form that has no search engine specified. These results are for Tomcat 4.1 and Resin 4.0; results will vary slightly among servers and will incorrectly omit the "Missing search string" message in JRun 4. In Internet Explorer, you must modify the browser settings as described in the previous section (see the 404 entry) to see the error message.

Listing 6.5 SearchEngineForm.java

```
package coreservlets;

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

/** Servlet that builds the HTML form that gathers input
 *  for the search engine servlet. This servlet first
 *  displays a textfield for the search query, then looks up
 *  the search engine names known to SearchUtilities and
 *  displays a list of radio buttons, one for each search
 *  engine.
 */

public class SearchEngineForm extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "One-Stop Web Search!";
    String actionURL = "/servlet/coreservlets.SearchEngines";
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    out.println
      (docType +
       "<HTML>\n" +
       "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
       "<BODY BGCOLOR=\"#FDF5E6\">\n" +
       "<CENTER>\n" +
       "<H1>" + title + "</H1>\n" +
       "<FORM ACTION=\"" + actionURL + "\">\n" +
       "  Search keywords: \n" +
       "  <INPUT TYPE=\"TEXT\" NAME=\"searchString\"><P>\n");
    SearchSpec[] specs = SearchUtilities.getCommonSpecs();
    for(int i=0; i<specs.length; i++) {
      String searchEngineName = specs[i].getName();
      out.println("<INPUT TYPE=\"RADIO\" " +
                  "NAME=\"searchEngine\" " +
                  "VALUE=\"" + searchEngineName + "\">\n");
      out.println(searchEngineName + "<BR>\n");
    }
    out.println
      ("<BR>  <INPUT TYPE=\"SUBMIT\">\n" +
       "</FORM>\n" +
       "</CENTER></BODY></HTML>");
  }
}
```

Generating the Server Response: HTTP Response Headers

Topics in This Chapter

- Format of the HTTP response
- Setting response headers
- Understanding what response headers are good for
- Building Excel spread sheets
- Generating JPEG images dynamically
- Sending incremental updates to the browser

Chapter 7

As discussed in the previous chapter, 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 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 chapter. 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 chapter shows how to generate response headers, explains what the various headers are used for, and gives several examples.

7.1 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.

- **`setHeader(String headerName, String headerValue)`**
  This method sets the response header with the designated name to the given value.

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. The methods `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(String mimeType)`**
  This method sets the `Content-Type` header and is used by the majority of servlets.

- **`setContentLength(int length)`**
  This method sets the `Content-Length` header, which is useful if the browser supports persistent (keep-alive) HTTP connections.
- **`addCookie(Cookie c)`**
  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 Chapter 8 (Handling Cookies) for a discussion of cookies.
- **`sendRedirect(String address)`**
  As discussed in the previous chapter, the `sendRedirect` method sets the `Location` header as well as setting the status code to 302. See Sections 6.3 (A Servlet That Redirects Users to Browser-Specific Pages) and 6.4 (A Front End to Various Search Engines) for examples.

7.2 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/ 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 available only in HTTP 1.1, especially if your servlet needs to run on the WWW "at large" rather than on an intranet—some older browsers support only HTTP 1.0. It is best to explicitly check the HTTP version with `request.getRequestProtocol` before using HTTP-1.1-specific 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 stipulate 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.

```
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-Disposition

The `Content-Disposition` header lets you request that the browser ask the user to save the response to disk in a file of the given name. It is used as follows:

```
Content-Disposition: attachment; filename=some-file-name
```

This header is particularly useful when you send the client non-HTML responses (e.g., Excel spreadsheets as in Section 7.3 or JPEG images as in Section 7.5). `Content-Disposition` was not part of the original HTTP specification; it was defined later in RFC 2183. Recall that you can download RFCs by going to http://rfc-editor.org/ and following the instructions.

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 5.4 (Sending Compressed Web Pages).

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 them, your servlet should write the document into a `ByteArrayOutputStream`, look up its size when done, put that into the `Content-Length` field with `response.setContentLength`, then send the content by `byteArrayStream.writeTo(response.getOutputStream())`.

Content-Type

The `Content-Type` header gives the MIME (Multipurpose Internet Mail Extension) type of the response document. Setting this header is so common that there is a special method in `HttpServletResponse` for it: `setContentType`. MIME types are of the form *`maintype/subtype`* for officially registered types and of the form *`maintype`*`/x-`*`subtype`* for unregistered types. Most servlets specify `text/html`; they can, however, specify other types instead. This is important partly because servlets directly generate other MIME types (as in the Excel and JPEG examples of this chapter), but also partly because servlets are used as the glue to connect other applications to the Web. OK, so you have Adobe Acrobat to generate PDF, GhostScript to generate PostScript, and a database application to search indexed MP3 files. But you still need a servlet to answer the HTTP request, invoke the helper application, and set the `Content-Type` header, even though the servlet probably simply passes the output of the helper application directly to the client.

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.

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

Table 7.1 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/ 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. For common unregistered types, http://www.ltsw.se/knbase/internet/mime.htp is a good source.

Table 7.1 Common MIME Types

Type	Meaning
application/msword	Microsoft Word document
application/octet-stream	Unrecognized or binary data
application/pdf	Acrobat (.pdf) file
application/postscript	PostScript file

Table 7.1 Common MIME Types *(continued)*

Type	Meaning
application/vnd.lotus-notes	Lotus Notes file
application/vnd.ms-excel	Excel spreadsheet
application/vnd.ms-powerpoint	PowerPoint presentation
application/x-gzip	Gzip archive
application/x-java-archive	JAR file
application/x-java-serialized-object	Serialized Java object
application/x-java-vm	Java bytecode (`.class`) file
application/zip	Zip archive
audio/basic	Sound file in .au or .snd format
audio/midi	MIDI sound file
audio/x-aiff	AIFF sound file
audio/x-wav	Microsoft Windows sound file
image/gif	GIF image
image/jpeg	JPEG image
image/png	PNG image
image/tiff	TIFF image
image/x-xbitmap	X Windows bitmap image
text/css	HTML cascading style sheet
text/html	HTML document
text/plain	Plain text
text/xml	XML
video/mpeg	MPEG video clip
video/quicktime	QuickTime video clip

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 header 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. However, some browsers ignore dates before January 1, 1980, so do not use 0 as the value of the `Expires` header.

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

```
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.

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 the lottery number servlet of Section 3.6, "The Servlet Life Cycle") 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`. See Sections 6.3 (A Servlet That Redirects Users to Browser-Specific Pages) and 6.4 (A Front End to Various Search Engines) for examples.

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

```
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 7.4 (Persistent Servlet State and Auto-Reloading Pages).

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" on which 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 Chapter 8 (Handling 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. For examples of the use of `WWW-Authenticate` and a discussion of the various security mechanisms available to servlets and JSP pages, see the chapters on Web application security in Volume 2 of this book.

7.3 Building Excel Spreadsheets

Although servlets usually generate HTML output, they are not required to do so. HTTP is fundamental to servlets; HTML is not. Now, it is sometimes useful to generate Microsoft Excel content so that users can save the results in a report and so that you can make use of the built-in formula support in Excel. Excel accepts input in at least three distinct formats: tab-separated data, HTML tables, and a native binary format.

In this section, we illustrate the use of tab-separated data to generate spreadsheets. In Chapter 12 (Controlling the Structure of Generated Servlets: The JSP page Directive), we show how to build Excel spreadsheets by using HTML-table format. No matter the format, the key is to use the `Content-Type` response header to tell the client that you are sending a spreadsheet. You use the shorthand `setContentType` method to set the `Content-Type` header, and the MIME type for Excel spreadsheets is `application/vnd.ms-excel`. So, to generate Excel spreadsheets, just do:

```
response.setContentType("application/vnd.ms-excel");
PrintWriter out = response.getWriter();
```

Then, simply print some entries with tabs (`\t` in Java strings) in between. That's it: no `DOCTYPE`, no `HEAD`, no `BODY`: those are all HTML-specific things.

Listing 7.1 presents a simple servlet that builds an Excel spreadsheet that compares apples and oranges. Note that `=SUM(col:col)` sums a range of columns in Excel. Figure 7–1 shows the results.

Listing 7.1 ApplesAndOranges.java

```
package coreservlets;

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

/** Servlet that creates Excel spreadsheet comparing
 *  apples and oranges.
 */

public class ApplesAndOranges extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
      response.setContentType("application/vnd.ms-excel");
      PrintWriter out = response.getWriter();
      out.println("\tQ1\tQ2\tQ3\tQ4\tTotal");
      out.println("Apples\t78\t87\t92\t29\t=SUM(B2:E2)");
      out.println("Oranges\t77\t86\t93\t30\t=SUM(B3:E3)");
  }
}
```

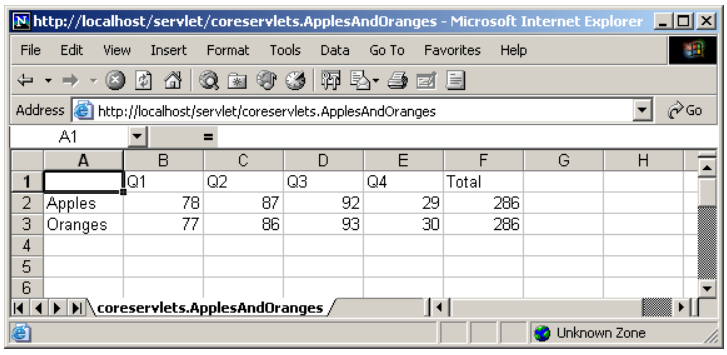


Figure 7–1 Result of the `ApplesAndOranges` servlet in Internet Explorer on a system that has Microsoft Office installed.

7.4 Persistent Servlet State and Auto-Reloading Pages

Suppose your servlet or JSP page performs a calculation that takes a long time to complete: say, 20 seconds or more. In such a case, it is not reasonable to complete the computation and then send the results to the client—by that time the client may have given up and left the page or, worse, have hit the Reload button and restarted the process. To deal with requests that take a long time to process (or whose results periodically change), you need the following capabilities:

- **A way to store data between requests.** For data that is not specific to any one client, store it in a field (instance variable) of the servlet. For data that is specific to a user, store it in the `HttpSession` object (see Chapter 9, "Session Tracking"). For data that needs to be available to other servlets or JSP pages, store it in the `ServletContext` (see the section on sharing data in Chapter 14, "Using JavaBeans Components in JSP Documents").
- **A way to keep computations running after the response is sent to the user.** This task is simple: just start a `Thread`. The thread started by the system to answer requests automatically finishes when the response is finished, but other threads can keep running. The only subtlety: set the thread priority to a low value so that you do not slow down the server.
- **A way to get the updated results to the browser when they are ready.** Unfortunately, because browsers do not maintain an open connection to the server, there is no easy way for the server to proactively send the new results to the browser. Instead, the browser needs to be told to ask for updates. That is the purpose of the `Refresh` response header.

Finding Prime Numbers for Use with Public Key Cryptography

Here is an example that lets you ask for a list of some large, randomly chosen prime numbers. As you are probably aware, access to large prime numbers is the key to most public-key cryptography systems, the kind of encryption systems used on the Web (e.g., for SSL and X509 certificates). Finding prime numbers may take some time for very large numbers (e.g., 100 digits), so the servlet immediately returns

initial results but then keeps calculating, using a low-priority thread so that it won't degrade Web server performance. If the calculations are not complete, the servlet instructs the browser to ask for a new page in a few seconds by sending it a `Refresh` header.

In addition to illustrating the value of HTTP response headers (`Refresh` in this case), this example shows two other valuable servlet capabilities. First, it shows that the same servlet can handle multiple simultaneous connections, each with its own thread. So, while one thread is finishing a calculation for one client, another client can connect and still see partial results.

Second, this example shows how easy it is for servlets to maintain state between requests, something that is cumbersome to implement in most competing technologies (even .NET, which is perhaps the best of the alternatives). Only a single instance of the servlet is created, and each request simply results in a new thread calling the servlet's `service` method (which calls `doGet` or `doPost`). So, shared data simply has to be placed in a regular instance variable (field) of the servlet. Thus, the servlet can access the appropriate ongoing calculation when the browser reloads the page and can keep a list of the *N* most recently requested results, returning them immediately if a new request specifies the same parameters as a recent one. Of course, the normal rules that require authors to synchronize multithreaded access to shared data still apply to servlets. Servlets can also store persistent data in the `ServletContext` object that is available through the `getServletContext` method. `ServletContext` has `setAttribute` and `getAttribute` methods that let you store arbitrary data associated with specified keys. The difference between storing data in instance variables and storing it in the `ServletContext` is that the `ServletContext` is shared by all servlets and JSP pages in the Web application.

Listing 7.2 shows the main servlet class. First, it receives a request that specifies two parameters: `numPrimes` and `numDigits`. These values are normally collected from the user and sent to the servlet by means of a simple HTML form. Listing 7.3 shows the source code and Figure 7–2 shows the result. Next, these parameters are converted to integers by means of a simple utility that uses `Integer.parseInt` (see Listing 7.6). These values are then matched by the `findPrimeList` method to an `ArrayList` of recent or ongoing calculations to see if a previous computation corresponds to the same two values. If so, that previous value (of type `PrimeList`) is used; otherwise, a new `PrimeList` is created and stored in the ongoing-calculations `Vector`, potentially displacing the oldest previous list. Next, that `PrimeList` is checked to determine whether it has finished finding all of its primes. If not, the client is sent a `Refresh` header to tell it to come back in five seconds for updated results. Either way, a bulleted list of the current values is returned to the client. See Figures 7–3 through 7–5 for representative results.

Listing 7.2 PrimeNumberServlet.java

```
package coreservlets;

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

/** Servlet that processes a request to generate n
 *  prime numbers, each with at least m digits.
 *  It performs the calculations in a low-priority background
 *  thread, returning only the results it has found so far.
 *  If these results are not complete, it sends a Refresh
 *  header instructing the browser to ask for new results a
 *  little while later. It also maintains a list of a
 *  small number of previously calculated prime lists
 *  to return immediately to anyone who supplies the
 *  same n and m as a recently completed computation.
 */

public class PrimeNumberServlet extends HttpServlet {
  private ArrayList primeListCollection = new ArrayList();
  private int maxPrimeLists = 30;

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    int numPrimes =
      ServletUtilities.getIntParameter(request,
                                       "numPrimes", 50);
    int numDigits =
      ServletUtilities.getIntParameter(request,
                                       "numDigits", 120);
    PrimeList primeList =
      findPrimeList(primeListCollection, numPrimes, numDigits);
    if (primeList == null) {
      primeList = new PrimeList(numPrimes, numDigits, true);
      // Multiple servlet request threads share the instance
      // variables (fields) of PrimeNumbers. So
      // synchronize all access to servlet fields.
      synchronized(primeListCollection) {
        if (primeListCollection.size() >= maxPrimeLists)
          primeListCollection.remove(0);
        primeListCollection.add(primeList);
      }
    }
    ArrayList currentPrimes = primeList.getPrimes();
    int numCurrentPrimes = currentPrimes.size();
    int numPrimesRemaining = (numPrimes - numCurrentPrimes);
```

Listing 7.2 PrimeNumberServlet.java *(continued)*

```
    boolean isLastResult = (numPrimesRemaining == 0);
    if (!isLastResult) {
      response.setIntHeader("Refresh", 5);
    }
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Some " + numDigits + "-Digit Prime Numbers";
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H2 ALIGN=CENTER>" + title + "</H2>\n" +
                "<H3>Primes found with " + numDigits +
                " or more digits: " + numCurrentPrimes +
                ".</H3>");
    if (isLastResult)
      out.println("<B>Done searching.</B>");
    else
      out.println("<B>Still looking for " + numPrimesRemaining +
                  " more<BLINK>...</BLINK></B>");
    out.println("<OL>");
    for(int i=0; i<numCurrentPrimes; i++) {
      out.println("  <LI>" + currentPrimes.get(i));
    }
    out.println("</OL>");
    out.println("</BODY></HTML>");
  }

  // See if there is an existing ongoing or completed
  // calculation with the same number of primes and number
  // of digits per prime. If so, return those results instead
  // of starting a new background thread. Keep this list
  // small so that the Web server doesn't use too much memory.
  // Synchronize access to the list since there may be
  // multiple simultaneous requests.

  private PrimeList findPrimeList(ArrayList primeListCollection,
                                  int numPrimes,
                                  int numDigits) {
      for(int i=0; i<primeListCollection.size(); i++) {
        PrimeList primes =
          (PrimeList)primeListCollection.get(i);
    synchronized(primeListCollection) {
        if ((numPrimes == primes.numPrimes()) &&
            (numDigits == primes.numDigits()))
          return(primes);
      }
      return(null);
    }
  }
}
```

Listing 7.3 PrimeNumbers.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Finding Large Prime Numbers</TITLE>
</HEAD>
<BODY BGCOLOR="#FDF5E6">
<CENTER>
<H2>Finding Large Prime Numbers</H2>
<BR><BR>
<FORM ACTION="/servlet/coreservlets.PrimeNumberServlet">
  <B>Number of primes to calculate:</B>
  <INPUT TYPE="TEXT" NAME="numPrimes" VALUE=25 SIZE=4><BR>
  <B>Number of digits:</B>
  <INPUT TYPE="TEXT" NAME="numDigits" VALUE=150 SIZE=3><BR>
  <INPUT TYPE="SUBMIT" VALUE="Start Calculating">
</FORM>
</CENTER>
</BODY></HTML>
```

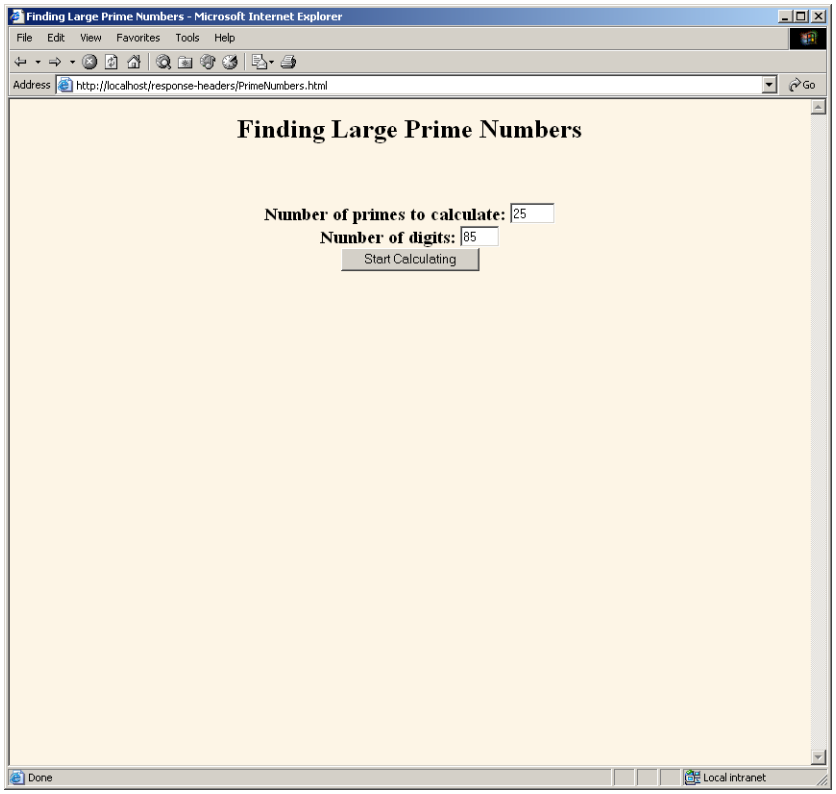


Figure 7–2 Front end to the prime-number-generation servlet.

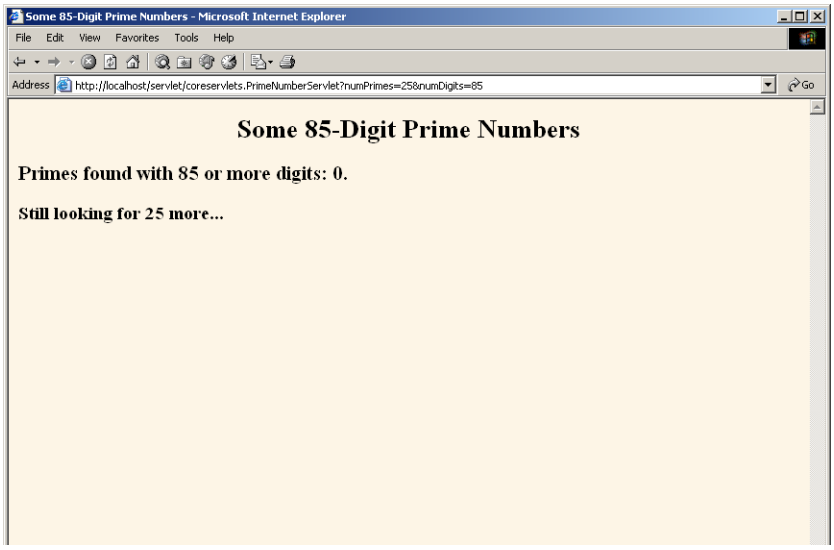


Figure 7–3 Initial results of the prime-number-generation servlet. A quick result is sent to the browser, along with instructions (in the `Refresh` header) to reconnect for an update in five seconds.

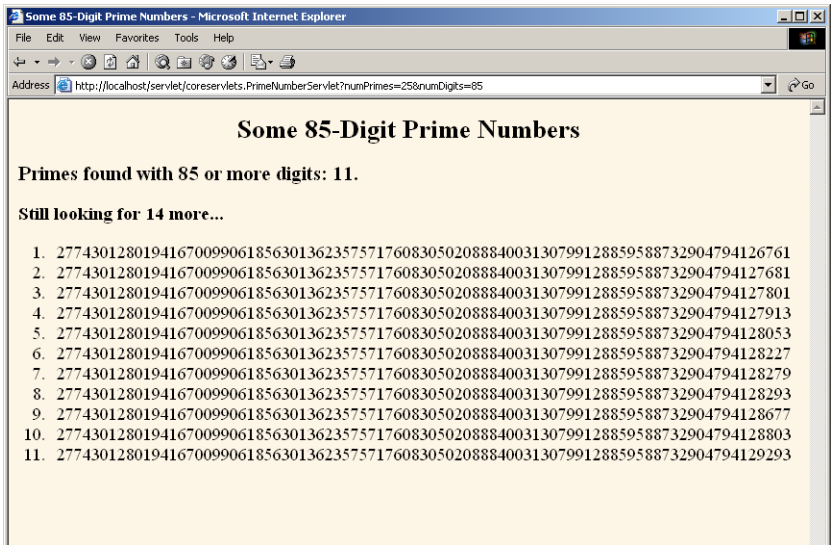


Figure 7–4 Intermediate results of the prime-number-generation servlet. The servlet stores the previous computations and matches the current request with the stored values by comparing the request parameters (the size and number of primes to compute). Other clients that request the same parameters see the same already computed results.

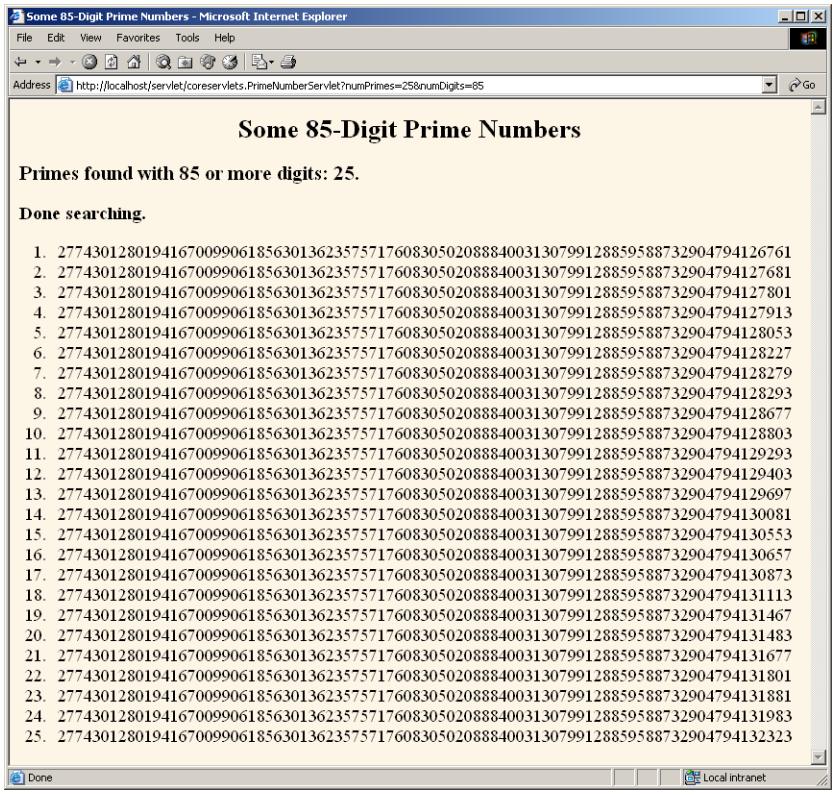


Figure 7–5 Final results of the prime-number-generation servlet. Since the servlet has computed as many primes as the user requested, no `Refresh` header is sent to the browser and the page is no longer reloaded automatically.

Listings 7.4 (PrimeList.java) and 7.5 (Primes.java) present auxiliary code used by the servlet. PrimeList.java handles the background thread for the creation of a list of primes for a specific set of values. The point of this example is twofold: that servlets can maintain data between requests by storing it in instance variables (or the `ServletContext`) and that the servlet can use the `Refresh` header to instruct the browser to return for updates. However, if you care about the gory details of prime-number generation, Primes.java contains the low-level algorithms for choosing a random number of a specified length and then finding a prime at or above that value. It uses built-in methods in the `BigInteger` class; the algorithm for determining if the number is prime is a probabilistic one and thus has a chance of being mistaken. However, the probability of being wrong can be specified, and we use an error value of 100. Assuming that the algorithm used in most Java implementations is the

Miller-Rabin test, the likelihood of falsely reporting a composite (i.e., non-prime) number as prime is provably less than 2^{100}. This is almost certainly smaller than the likelihood of a hardware error or random radiation causing an incorrect response in a deterministic algorithm, and thus the algorithm can be considered deterministic.

Listing 7.4 PrimeList.java

```
package coreservlets;

import java.util.*;
import java.math.BigInteger;

/** Creates an ArrayList of large prime numbers, usually in
 *  a low-priority background thread. Provides a few small
 *  thread-safe access methods.
 */

public class PrimeList implements Runnable {
  private ArrayList primesFound;
  private int numPrimes, numDigits;

  /** Finds numPrimes prime numbers, each of which is
   *  numDigits long or longer. You can set it to return
   *  only when done, or have it return immediately,
   *  and you can later poll it to see how far it
   *  has gotten.
   */

  public PrimeList(int numPrimes, int numDigits,
                   boolean runInBackground) {
    primesFound = new ArrayList(numPrimes);
    this.numPrimes = numPrimes;
    this.numDigits = numDigits;
    if (runInBackground) {
      Thread t = new Thread(this);
      // Use low priority so you don't slow down server.
      t.setPriority(Thread.MIN_PRIORITY);
      t.start();
    } else {
      run();
    }
  }
```

Listing 7.4 PrimeList.java *(continued)*

```
  public void run() {
    BigInteger start = Primes.random(numDigits);
    for(int i=0; i<numPrimes; i++) {
      start = Primes.nextPrime(start);
      synchronized(this) {
        primesFound.add(start);
      }
    }
  }

  public synchronized boolean isDone() {
    return(primesFound.size() == numPrimes);
  }

  public synchronized ArrayList getPrimes() {
    if (isDone())
      return(primesFound);
    else
      return((ArrayList)primesFound.clone());
  }

  public int numDigits() {
    return(numDigits);
  }

  public int numPrimes() {
    return(numPrimes);
  }

  public synchronized int numCalculatedPrimes() {
    return(primesFound.size());
  }
}
```

Listing 7.5 Primes.java

```
package coreservlets;

import java.math.BigInteger;

/** A few utilities to generate a large random BigInteger,
 *  and find the next prime number above a given BigInteger.
 */

public class Primes {
  // Note that BigInteger.ZERO and BigInteger.ONE are
  // unavailable in JDK 1.1.
  private static final BigInteger ZERO = BigInteger.ZERO;
  private static final BigInteger ONE = BigInteger.ONE;
  private static final BigInteger TWO = new BigInteger("2");

  // Likelihood of false prime is less than 1/2^ERR_VAL.
  // Presumably BigInteger uses the Miller-Rabin test or
  // equivalent, and thus is NOT fooled by Carmichael numbers.
  // See section 33.8 of Cormen et al.'s Introduction to
  // Algorithms for details.
  private static final int ERR_VAL = 100;

  public static BigInteger nextPrime(BigInteger start) {
    if (isEven(start))
      start = start.add(ONE);
    else
      start = start.add(TWO);
    if (start.isProbablePrime(ERR_VAL))
      return(start);
    else
      return(nextPrime(start));
  }

  private static boolean isEven(BigInteger n) {
    return(n.mod(TWO).equals(ZERO));
  }

  private static StringBuffer[] digits =
    { new StringBuffer("0"), new StringBuffer("1"),
      new StringBuffer("2"), new StringBuffer("3"),
      new StringBuffer("4"), new StringBuffer("5"),
      new StringBuffer("6"), new StringBuffer("7"),
      new StringBuffer("8"), new StringBuffer("9") };
```

Listing 7.5 Primes.java *(continued)*

```
  private static StringBuffer randomDigit(boolean isZeroOK) {
    int index;
    if (isZeroOK) {
      index = (int)Math.floor(Math.random() * 10);
    } else {
      index = 1 + (int)Math.floor(Math.random() * 9);
    }
    return(digits[index]);
  }

  /** Create a random big integer where every digit is
   *  selected randomly (except that the first digit
   *  cannot be a zero).
   */

  public static BigInteger random(int numDigits) {
    StringBuffer s = new StringBuffer("");
    for(int i=0; i<numDigits; i++) {
      if (i == 0) {
        // First digit must be non-zero.
        s.append(randomDigit(false));
      } else {
        s.append(randomDigit(true));
      }
    }
    return(new BigInteger(s.toString()));
  }

  /** Simple command-line program to test. Enter number
   *  of digits, and the program picks a random number of that
   *  length and then prints the first 50 prime numbers
   *  above that.
   */

  public static void main(String[] args) {
    int numDigits;
    try {
      numDigits = Integer.parseInt(args[0]);
    } catch (Exception e) { // No args or illegal arg.
      numDigits = 150;
    }
    BigInteger start = random(numDigits);
    for(int i=0; i<50; i++) {
      start = nextPrime(start);
      System.out.println("Prime " + i + " = " + start);
    }
  }
}
```

Listing 7.6 ServletUtilities.java (Excerpt)

```
package coreservlets;

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

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

public class ServletUtilities {
  // ...

  /** Read a parameter with the specified name, convert it
   *  to an int, and return it. Return the designated default
   *  value if the parameter doesn't exist or if it is an
   *  illegal integer format.
   */

  public static int getIntParameter(HttpServletRequest request,
                                    String paramName,
                                    int defaultValue) {
    String paramString = request.getParameter(paramName);
    int paramValue;
    try {
      paramValue = Integer.parseInt(paramString);
    } catch(NumberFormatException nfe) { // null or bad format
      paramValue = defaultValue;
    }
    return(paramValue);
  }
}
```

7.5 Using Servlets to Generate JPEG Images

Although servlets often generate HTML output, they certainly don't *always* do so. For example, Section 7.3 (Building Excel Spreadsheets) shows a servlet that builds Excel spreadsheets and returns them to the client. Here, we show you how to generate JPEG images.

First, let us summarize the two main steps servlets have to perform to build multimedia content.

1. **Inform the browser of the content type they are sending.** To accomplish this task, servlets set the `Content-Type` response header by using the `setContentType` method of `HttpServletResponse`.
2. **Send the output in the appropriate format.** This format varies among document types, of course, but in most cases you send binary data, not strings as you do with HTML documents. Consequently, servlets will usually get the raw output stream by using the `getOutputStream` method, rather than getting a `PrintWriter` by using `getWriter`.

Putting these two steps together, servlets that generate non-HTML content usually have a section of their `doGet` or `doPost` method that looks like this:

```
response.setContentType("type/subtype");
OutputStream out = response.getOutputStream();
```

Those are the two general steps required to build non-HTML content. Next, let's look at the specific steps required to generate JPEG images.

1. **Create a `BufferedImage`.**
   You create a `java.awt.image.BufferedImage` object by calling the `BufferedImage` constructor with a width, a height, and an image representation type as defined by one of the constants in the `BufferedImage` class. The representation type is not important, since we do not manipulate the bits of the `BufferedImage` directly and since most types yield identical results when converted to JPEG. We use `TYPE_INT_RGB`. Putting this all together, here is the normal process:

   ```
   int width = ...;
   int height = ...;
   BufferedImage image =
     new BufferedImage(width, height,
                       BufferedImage.TYPE_INT_RGB);
   ```

2. **Draw into the `BufferedImage`.**
   You accomplish this task by calling the image's `getGraphics` method, casting the resultant `Graphics` object to `Graphics2D`, then making use of Java 2D's rich set of drawing operations, coordinate transformations, font settings, and fill patterns to perform the drawing. Here is a simple example.

   ```
   Graphics2D g2d = (Graphics2D)image.getGraphics();
   g2d.setXxx(...);
   g2d.fill(someShape);
   g2d.draw(someShape);
   ```

3. **Set the `Content-Type` response header.**
   As already discussed, you use the `setContentType` method of `HttpServletResponse` for this task. The MIME type for JPEG images is `image/jpeg`. Thus, the code is as follows.

   ```
   response.setContentType("image/jpeg");
   ```

4. **Get an output stream.**
   As discussed previously, if you are sending binary data, you should call the `getOutputStream` method of `HttpServletResponse` rather than the `getWriter` method. For instance:

   ```
   OutputStream out = response.getOutputStream();
   ```

5. **Send the `BufferedImage` in JPEG format to the output stream.**
   Before JDK 1.4, accomplishing this task yourself required quite a bit of work. So, most people used a third-party utility for this purpose. In JDK 1.4 and later, however, the `ImageIO` class greatly simplifies this task. If you are using an application server that supports J2EE 1.4 (which includes servlets 2.4 and JSP 2.0), you are guaranteed to have JDK 1.4 or later. However, standalone servers are not absolutely required to use JDK 1.4, so be aware that this code depends on the Java version. When you use the `ImageIO` class, you just pass a `BufferedImage`, an image format type (`"jpg"`, `"png"`, etc.—call `ImageIO.getWriterFormatNames` for a complete list), and either an `OutputStream` or a `File` to the `write` method of `ImageIO`. Except for catching the required `IOException`, that's it! For example:

   ```
   try {
     ImageIO.write(image, "jpg", out);
   } catch(IOException ioe) {
     System.err.println("Error writing JPEG file: " + ioe);
   }
   ```

Listing 7.7 shows a servlet that reads `message`, `fontName`, and `fontSize` parameters and passes them to the `MessageImage` utility (Listing 7.8) to create a JPEG image showing the message in the designated face and size, with a gray, oblique-shadowed version of the message shown behind the main string. If the user presses the Show Font List button, then instead of building an image, the servlet displays a list of font names available on the server.

Listing 7.7 ShadowedText.java

```
package coreservlets;

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

/** Servlet that generates JPEG images representing
 *  a designated message with an oblique-shadowed
 *  version behind it.
 */

public class ShadowedText extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    String wantsList = request.getParameter("showList");
    if (wantsList != null) {
      showFontList(response);
    } else {
      String message = request.getParameter("message");
      if ((message == null) || (message.length() == 0)) {
        message = "Missing 'message' parameter";
      }
      String fontName = request.getParameter("fontName");
      if ((fontName == null) || (fontName.length() == 0)) {
        fontName = "Serif";
      }
      String fontSizeString = request.getParameter("fontSize");
      int fontSize;
      try {
        fontSize = Integer.parseInt(fontSizeString);
      } catch(NumberFormatException nfe) {
        fontSize = 90;
      }
      response.setContentType("image/jpeg");
      MessageImage.writeJPEG
        (MessageImage.makeMessageImage(message,
                                       fontName,
                                       fontSize),
         response.getOutputStream());
    }
  }
```

Listing 7.7 ShadowedText.java *(continued)*

```
  private void showFontList(HttpServletResponse response)
      throws IOException {
    PrintWriter out = response.getWriter();
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    String title = "Fonts Available on Server";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                "<UL>");
    String[] fontNames = MessageImage.getFontNames();
    for(int i=0; i<fontNames.length; i++) {
      out.println("  <LI>" + fontNames[i]);
    }
    out.println("</UL>\n" +
                "</BODY></HTML>");
  }
}
```

Listing 7.8 MessageImage.java

```
package coreservlets;

import java.awt.*;
import java.awt.geom.*;
import java.awt.image.*;
import java.io.*;
import javax.imageio.*;

/** Utilities for building images showing shadowed messages.
 *  <P>
 *  Requires JDK 1.4 since it uses the ImageIO class.
 *  JDK 1.4 is standard with J2EE-compliant app servers
 *  with servlets 2.4 and JSP 2.0. However, standalone
 *  servlet/JSP engines require only JDK 1.3 or later, and
 *  version 2.3 of the servlet spec requires only JDK
 *  1.2 or later. So, although most servers run on JDK 1.4,
 *  this code is not necessarily portable across all servers.
 */
```

Listing 7.8 MessageImage.java *(continued)*

```
public class MessageImage {

  /** Creates an Image of a string with an oblique
   *  shadow behind it. Used by the ShadowedText servlet.
   */

  public static BufferedImage makeMessageImage(String message,
                                               String fontName,
                                               int fontSize) {

    Font font = new Font(fontName, Font.PLAIN, fontSize);
    FontMetrics metrics = getFontMetrics(font);
    int messageWidth = metrics.stringWidth(message);
    int baselineX = messageWidth/10;
    int width = messageWidth+2*(baselineX + fontSize);
    int height = fontSize*7/2;
    int baselineY = height*8/10;
    BufferedImage messageImage =
      new BufferedImage(width, height,
                        BufferedImage.TYPE_INT_RGB);
    Graphics2D g2d = (Graphics2D)messageImage.getGraphics();
    g2d.setBackground(Color.white);
    g2d.clearRect(0, 0, width, height);
    g2d.setFont(font);
    g2d.translate(baselineX, baselineY);
    g2d.setPaint(Color.lightGray);
    AffineTransform origTransform = g2d.getTransform();
    g2d.shear(-0.95, 0);
    g2d.scale(1, 3);
    g2d.drawString(message, 0, 0);
    g2d.setTransform(origTransform);
    g2d.setPaint(Color.black);
    g2d.drawString(message, 0, 0);
    return(messageImage);
  }

  public static void writeJPEG(BufferedImage image,
                               OutputStream out) {
    try {
      ImageIO.write(image, "jpg", out);
    } catch(IOException ioe) {
      System.err.println("Error outputting JPEG: " + ioe);
    }
  }
```

Listing 7.8 MessageImage.java *(continued)*

```
  public static void writeJPEG(BufferedImage image,
                               File file) {
    try {
      ImageIO.write(image, "jpg", file);
    } catch(IOException ioe) {
      System.err.println("Error writing JPEG file: " + ioe);
    }
  }

  public static String[] getFontNames() {
    GraphicsEnvironment env =
      GraphicsEnvironment.getLocalGraphicsEnvironment();
    return(env.getAvailableFontFamilyNames());
  }

  /** We need a Graphics object to get a FontMetrics object
   *  (an object that says how big strings are in given fonts).
   *  But, you need an image from which to derive the Graphics
   *  object. Since the size of the "real" image will depend on
   *  how big the string is, we create a very small temporary
   *  image first, get the FontMetrics, figure out how
   *  big the real image should be, then use a real image
   *  of that size.
   */

  private static FontMetrics getFontMetrics(Font font) {
    BufferedImage tempImage =
      new BufferedImage(1, 1, BufferedImage.TYPE_INT_RGB);
    Graphics2D g2d = (Graphics2D)tempImage.getGraphics();
    return(g2d.getFontMetrics(font));
  }
}
```

Listing 7.9 (Figure 7–6) shows an HTML form used as a front end to the servlet. Figures 7–7 through 7–10 show some possible results. Just to simplify experimentation, Listing 7.10 presents an interactive application that lets you specify the message and font name on the command line, outputting the image to a file.

Listing 7.9 ShadowedText.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>JPEG Generation Service</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN="CENTER">JPEG Generation Service</H1>
Welcome to the <I>free</I> trial edition of our JPEG
generation service. Enter a message, a font name,
and a font size below, then submit the form. You will
be returned a JPEG image showing the message in the
designated font, with an oblique "shadow" of the message
behind it. Once you get an image you are satisfied with,
right-click
on it (or click while holding down the SHIFT key) to save
it to your local disk.
<P>
The server is currently on Windows, so the font name must
be either a standard Java font name (e.g., Serif, SansSerif,
or Monospaced) or a Windows font name (e.g., Arial Black).
Unrecognized font names will revert to Serif. Press the
"Show Font List" button for a complete list.

<FORM ACTION="/servlet/coreservlets.ShadowedText">
  <CENTER>
    Message:
    <INPUT TYPE="TEXT" NAME="message"><BR>
    Font name:
    <INPUT TYPE="TEXT" NAME="fontName" VALUE="Serif"><BR>
    Font size:
    <INPUT TYPE="TEXT" NAME="fontSize" VALUE="90"><P>
    <INPUT TYPE="SUBMIT" VALUE="Build Image"><P>
    <INPUT TYPE="SUBMIT" NAME="showList" VALUE="Show Font List">
  </CENTER>
</FORM>

</BODY></HTML>
```

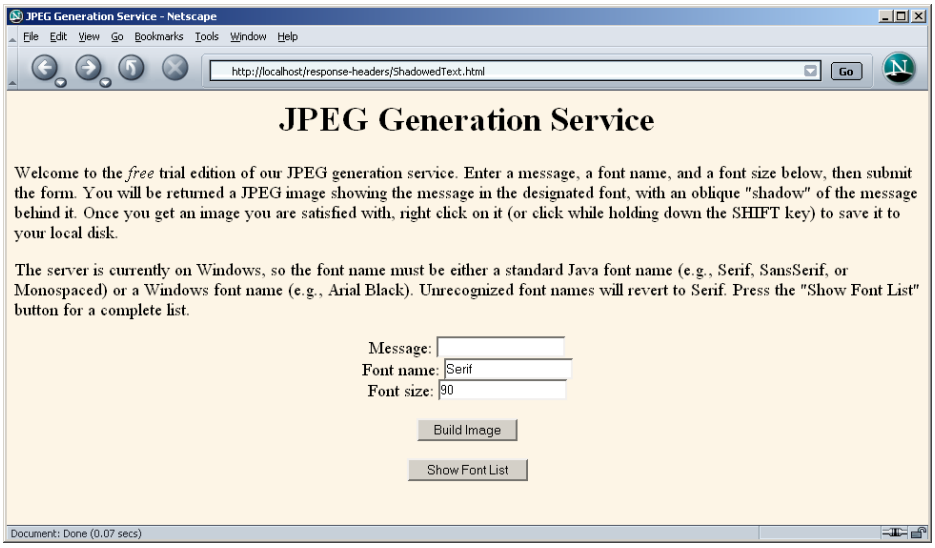


Figure 7–6 Front end to the image-generation servlet.

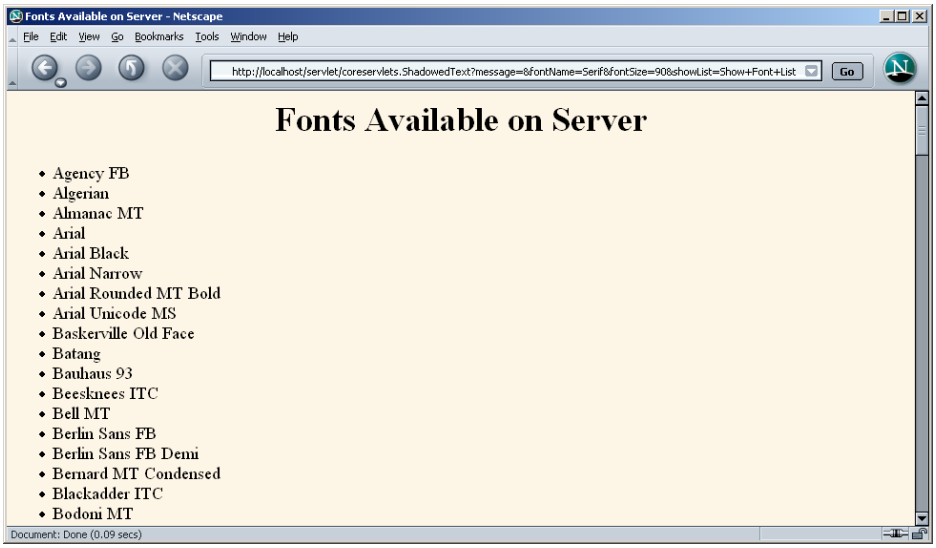


Figure 7–7 Result of servlet when the client selects Show Font List.

Figure 7–8 One possible result of the image-generation servlet. The client can save the image to disk as *somename*.jpg and use it in Web pages or other applications.

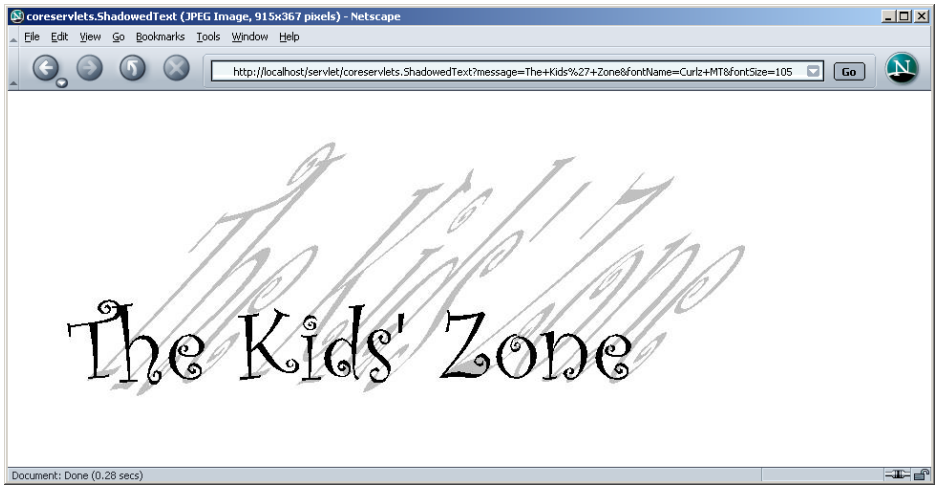


Figure 7–9 A second possible result of the image-generation servlet.

Figure 7–10 A third possible result of the image-generation servlet.

Listing 7.10 ImageTest.java

```
package coreservlets;

import java.io.*;

public class ImageTest {
  public static void main(String[] args) {
    String message = "Testing";
    String font = "Arial";
    if (args.length > 0) {
      message = args[0];
    }
    if (args.length > 1) {
      font = args[1];
    }
    MessageImage.writeJPEG
      (MessageImage.makeMessageImage(message, font, 40),
       new File("ImageTest.jpg"));
  }
}
```

Generating the Server Response: HTTP Response Headers

Topics in This Chapter

- Format of the HTTP response
- Setting response headers
- Understanding what response headers are good for
- Building Excel spread sheets
- Generating JPEG images dynamically
- Sending incremental updates to the browser

Chapter 7

As discussed in the previous chapter, 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 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 chapter. 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 chapter shows how to generate response headers, explains what the various headers are used for, and gives several examples.

7.1 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.

- **`setHeader(String headerName, String headerValue)`**
  This method sets the response header with the designated name to the given value.

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. The methods `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(String mimeType)`**
  This method sets the `Content-Type` header and is used by the majority of servlets.

- **`setContentLength(int length)`**
  This method sets the `Content-Length` header, which is useful if the browser supports persistent (keep-alive) HTTP connections.
- **`addCookie(Cookie c)`**
  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 Chapter 8 (Handling Cookies) for a discussion of cookies.
- **`sendRedirect(String address)`**
  As discussed in the previous chapter, the `sendRedirect` method sets the `Location` header as well as setting the status code to 302. See Sections 6.3 (A Servlet That Redirects Users to Browser-Specific Pages) and 6.4 (A Front End to Various Search Engines) for examples.

7.2 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/ 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 available only in HTTP 1.1, especially if your servlet needs to run on the WWW "at large" rather than on an intranet—some older browsers support only HTTP 1.0. It is best to explicitly check the HTTP version with `request.getRequestProtocol` before using HTTP-1.1-specific 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 stipulate 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.

```
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-Disposition

The `Content-Disposition` header lets you request that the browser ask the user to save the response to disk in a file of the given name. It is used as follows:

```
Content-Disposition: attachment; filename=some-file-name
```

This header is particularly useful when you send the client non-HTML responses (e.g., Excel spreadsheets as in Section 7.3 or JPEG images as in Section 7.5). `Content-Disposition` was not part of the original HTTP specification; it was defined later in RFC 2183. Recall that you can download RFCs by going to http://rfc-editor.org/ and following the instructions.

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 5.4 (Sending Compressed Web Pages).

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 them, your servlet should write the document into a `ByteArrayOutputStream`, look up its size when done, put that into the `Content-Length` field with `response.setContentLength`, then send the content by `byteArrayStream.writeTo(response.getOutputStream())`.

Content-Type

The `Content-Type` header gives the MIME (Multipurpose Internet Mail Extension) type of the response document. Setting this header is so common that there is a special method in `HttpServletResponse` for it: `setContentType`. MIME types are of the form *`maintype/subtype`* for officially registered types and of the form *`maintype`*`/x-`*`subtype`* for unregistered types. Most servlets specify `text/html`; they can, however, specify other types instead. This is important partly because servlets directly generate other MIME types (as in the Excel and JPEG examples of this chapter), but also partly because servlets are used as the glue to connect other applications to the Web. OK, so you have Adobe Acrobat to generate PDF, GhostScript to generate PostScript, and a database application to search indexed MP3 files. But you still need a servlet to answer the HTTP request, invoke the helper application, and set the `Content-Type` header, even though the servlet probably simply passes the output of the helper application directly to the client.

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.

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

Table 7.1 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/ 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. For common unregistered types, http://www.ltsw.se/knbase/internet/mime.htp is a good source.

Table 7.1 Common MIME Types

Type	Meaning
application/msword	Microsoft Word document
application/octet-stream	Unrecognized or binary data
application/pdf	Acrobat (.pdf) file
application/postscript	PostScript file

Table 7.1 Common MIME Types *(continued)*

Type	Meaning
application/vnd.lotus-notes	Lotus Notes file
application/vnd.ms-excel	Excel spreadsheet
application/vnd.ms-powerpoint	PowerPoint presentation
application/x-gzip	Gzip archive
application/x-java-archive	JAR file
application/x-java-serialized-object	Serialized Java object
application/x-java-vm	Java bytecode (`.class`) file
application/zip	Zip archive
audio/basic	Sound file in .au or .snd format
audio/midi	MIDI sound file
audio/x-aiff	AIFF sound file
audio/x-wav	Microsoft Windows sound file
image/gif	GIF image
image/jpeg	JPEG image
image/png	PNG image
image/tiff	TIFF image
image/x-xbitmap	X Windows bitmap image
text/css	HTML cascading style sheet
text/html	HTML document
text/plain	Plain text
text/xml	XML
video/mpeg	MPEG video clip
video/quicktime	QuickTime video clip

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 header 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. However, some browsers ignore dates before January 1, 1980, so do not use 0 as the value of the `Expires` header.

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

```
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.

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 the lottery number servlet of Section 3.6, "The Servlet Life Cycle") 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`. See Sections 6.3 (A Servlet That Redirects Users to Browser-Specific Pages) and 6.4 (A Front End to Various Search Engines) for examples.

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

```
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 7.4 (Persistent Servlet State and Auto-Reloading Pages).

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" on which 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 Chapter 8 (Handling 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. For examples of the use of `WWW-Authenticate` and a discussion of the various security mechanisms available to servlets and JSP pages, see the chapters on Web application security in Volume 2 of this book.

7.3 Building Excel Spreadsheets

Although servlets usually generate HTML output, they are not required to do so. HTTP is fundamental to servlets; HTML is not. Now, it is sometimes useful to generate Microsoft Excel content so that users can save the results in a report and so that you can make use of the built-in formula support in Excel. Excel accepts input in at least three distinct formats: tab-separated data, HTML tables, and a native binary format.

In this section, we illustrate the use of tab-separated data to generate spreadsheets. In Chapter 12 (Controlling the Structure of Generated Servlets: The JSP page Directive), we show how to build Excel spreadsheets by using HTML-table format. No matter the format, the key is to use the `Content-Type` response header to tell the client that you are sending a spreadsheet. You use the shorthand `setContentType` method to set the `Content-Type` header, and the MIME type for Excel spreadsheets is `application/vnd.ms-excel`. So, to generate Excel spreadsheets, just do:

```
response.setContentType("application/vnd.ms-excel");
PrintWriter out = response.getWriter();
```

Then, simply print some entries with tabs (`\t` in Java strings) in between. That's it: no `DOCTYPE`, no `HEAD`, no `BODY`: those are all HTML-specific things.

Listing 7.1 presents a simple servlet that builds an Excel spreadsheet that compares apples and oranges. Note that `=SUM(col:col)` sums a range of columns in Excel. Figure 7–1 shows the results.

Listing 7.1 ApplesAndOranges.java

```
package coreservlets;

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

/** Servlet that creates Excel spreadsheet comparing
 *  apples and oranges.
 */

public class ApplesAndOranges extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
      response.setContentType("application/vnd.ms-excel");
      PrintWriter out = response.getWriter();
      out.println("\tQ1\tQ2\tQ3\tQ4\tTotal");
      out.println("Apples\t78\t87\t92\t29\t=SUM(B2:E2)");
      out.println("Oranges\t77\t86\t93\t30\t=SUM(B3:E3)");
  }
}
```

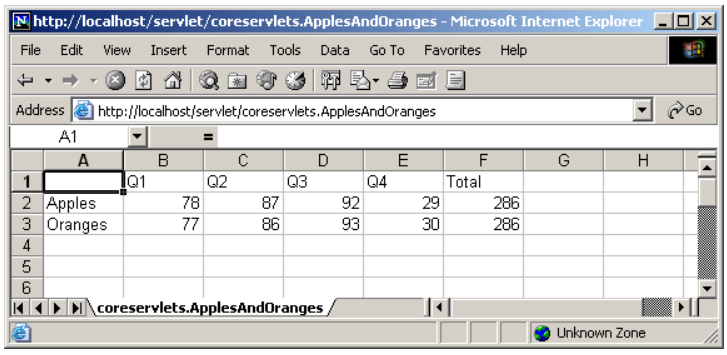


Figure 7–1 Result of the `ApplesAndOranges` servlet in Internet Explorer on a system that has Microsoft Office installed.

7.4 Persistent Servlet State and Auto-Reloading Pages

Suppose your servlet or JSP page performs a calculation that takes a long time to complete: say, 20 seconds or more. In such a case, it is not reasonable to complete the computation and then send the results to the client—by that time the client may have given up and left the page or, worse, have hit the Reload button and restarted the process. To deal with requests that take a long time to process (or whose results periodically change), you need the following capabilities:

- **A way to store data between requests.** For data that is not specific to any one client, store it in a field (instance variable) of the servlet. For data that is specific to a user, store it in the `HttpSession` object (see Chapter 9, "Session Tracking"). For data that needs to be available to other servlets or JSP pages, store it in the `ServletContext` (see the section on sharing data in Chapter 14, "Using JavaBeans Components in JSP Documents").
- **A way to keep computations running after the response is sent to the user.** This task is simple: just start a `Thread`. The thread started by the system to answer requests automatically finishes when the response is finished, but other threads can keep running. The only subtlety: set the thread priority to a low value so that you do not slow down the server.
- **A way to get the updated results to the browser when they are ready.** Unfortunately, because browsers do not maintain an open connection to the server, there is no easy way for the server to proactively send the new results to the browser. Instead, the browser needs to be told to ask for updates. That is the purpose of the `Refresh` response header.

Finding Prime Numbers for Use with Public Key Cryptography

Here is an example that lets you ask for a list of some large, randomly chosen prime numbers. As you are probably aware, access to large prime numbers is the key to most public-key cryptography systems, the kind of encryption systems used on the Web (e.g., for SSL and X509 certificates). Finding prime numbers may take some time for very large numbers (e.g., 100 digits), so the servlet immediately returns

initial results but then keeps calculating, using a low-priority thread so that it won't degrade Web server performance. If the calculations are not complete, the servlet instructs the browser to ask for a new page in a few seconds by sending it a `Refresh` header.

In addition to illustrating the value of HTTP response headers (`Refresh` in this case), this example shows two other valuable servlet capabilities. First, it shows that the same servlet can handle multiple simultaneous connections, each with its own thread. So, while one thread is finishing a calculation for one client, another client can connect and still see partial results.

Second, this example shows how easy it is for servlets to maintain state between requests, something that is cumbersome to implement in most competing technologies (even .NET, which is perhaps the best of the alternatives). Only a single instance of the servlet is created, and each request simply results in a new thread calling the servlet's `service` method (which calls `doGet` or `doPost`). So, shared data simply has to be placed in a regular instance variable (field) of the servlet. Thus, the servlet can access the appropriate ongoing calculation when the browser reloads the page and can keep a list of the *N* most recently requested results, returning them immediately if a new request specifies the same parameters as a recent one. Of course, the normal rules that require authors to synchronize multithreaded access to shared data still apply to servlets. Servlets can also store persistent data in the `ServletContext` object that is available through the `getServletContext` method. `ServletContext` has `setAttribute` and `getAttribute` methods that let you store arbitrary data associated with specified keys. The difference between storing data in instance variables and storing it in the `ServletContext` is that the `ServletContext` is shared by all servlets and JSP pages in the Web application.

Listing 7.2 shows the main servlet class. First, it receives a request that specifies two parameters: `numPrimes` and `numDigits`. These values are normally collected from the user and sent to the servlet by means of a simple HTML form. Listing 7.3 shows the source code and Figure 7–2 shows the result. Next, these parameters are converted to integers by means of a simple utility that uses `Integer.parseInt` (see Listing 7.6). These values are then matched by the `findPrimeList` method to an `ArrayList` of recent or ongoing calculations to see if a previous computation corresponds to the same two values. If so, that previous value (of type `PrimeList`) is used; otherwise, a new `PrimeList` is created and stored in the ongoing-calculations `Vector`, potentially displacing the oldest previous list. Next, that `PrimeList` is checked to determine whether it has finished finding all of its primes. If not, the client is sent a `Refresh` header to tell it to come back in five seconds for updated results. Either way, a bulleted list of the current values is returned to the client. See Figures 7–3 through 7–5 for representative results.

Listing 7.2 PrimeNumberServlet.java

```
package coreservlets;

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

/** Servlet that processes a request to generate n
 *  prime numbers, each with at least m digits.
 *  It performs the calculations in a low-priority background
 *  thread, returning only the results it has found so far.
 *  If these results are not complete, it sends a Refresh
 *  header instructing the browser to ask for new results a
 *  little while later. It also maintains a list of a
 *  small number of previously calculated prime lists
 *  to return immediately to anyone who supplies the
 *  same n and m as a recently completed computation.
 */

public class PrimeNumberServlet extends HttpServlet {
  private ArrayList primeListCollection = new ArrayList();
  private int maxPrimeLists = 30;

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    int numPrimes =
      ServletUtilities.getIntParameter(request,
                                       "numPrimes", 50);
    int numDigits =
      ServletUtilities.getIntParameter(request,
                                       "numDigits", 120);
    PrimeList primeList =
      findPrimeList(primeListCollection, numPrimes, numDigits);
    if (primeList == null) {
      primeList = new PrimeList(numPrimes, numDigits, true);
      // Multiple servlet request threads share the instance
      // variables (fields) of PrimeNumbers. So
      // synchronize all access to servlet fields.
      synchronized(primeListCollection) {
        if (primeListCollection.size() >= maxPrimeLists)
          primeListCollection.remove(0);
        primeListCollection.add(primeList);
      }
    }
    ArrayList currentPrimes = primeList.getPrimes();
    int numCurrentPrimes = currentPrimes.size();
    int numPrimesRemaining = (numPrimes - numCurrentPrimes);
```

Listing 7.2 PrimeNumberServlet.java *(continued)*

```
    boolean isLastResult = (numPrimesRemaining == 0);
    if (!isLastResult) {
      response.setIntHeader("Refresh", 5);
    }
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Some " + numDigits + "-Digit Prime Numbers";
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H2 ALIGN=CENTER>" + title + "</H2>\n" +
                "<H3>Primes found with " + numDigits +
                " or more digits: " + numCurrentPrimes +
                ".</H3>");
    if (isLastResult)
      out.println("<B>Done searching.</B>");
    else
      out.println("<B>Still looking for " + numPrimesRemaining +
                  " more<BLINK>...</BLINK></B>");
    out.println("<OL>");
    for(int i=0; i<numCurrentPrimes; i++) {
      out.println("  <LI>" + currentPrimes.get(i));
    }
    out.println("</OL>");
    out.println("</BODY></HTML>");
  }

  // See if there is an existing ongoing or completed
  // calculation with the same number of primes and number
  // of digits per prime. If so, return those results instead
  // of starting a new background thread. Keep this list
  // small so that the Web server doesn't use too much memory.
  // Synchronize access to the list since there may be
  // multiple simultaneous requests.

  private PrimeList findPrimeList(ArrayList primeListCollection,
                                  int numPrimes,
                                  int numDigits) {
      for(int i=0; i<primeListCollection.size(); i++) {
        PrimeList primes =
          (PrimeList)primeListCollection.get(i);
    synchronized(primeListCollection) {
        if ((numPrimes == primes.numPrimes()) &&
            (numDigits == primes.numDigits()))
          return(primes);
      }
      return(null);
    }
  }
}
```

Listing 7.3 PrimeNumbers.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Finding Large Prime Numbers</TITLE>
</HEAD>
<BODY BGCOLOR="#FDF5E6">
<CENTER>
<H2>Finding Large Prime Numbers</H2>
<BR><BR>
<FORM ACTION="/servlet/coreservlets.PrimeNumberServlet">
  <B>Number of primes to calculate:</B>
  <INPUT TYPE="TEXT" NAME="numPrimes" VALUE=25 SIZE=4><BR>
  <B>Number of digits:</B>
  <INPUT TYPE="TEXT" NAME="numDigits" VALUE=150 SIZE=3><BR>
  <INPUT TYPE="SUBMIT" VALUE="Start Calculating">
</FORM>
</CENTER>
</BODY></HTML>
```

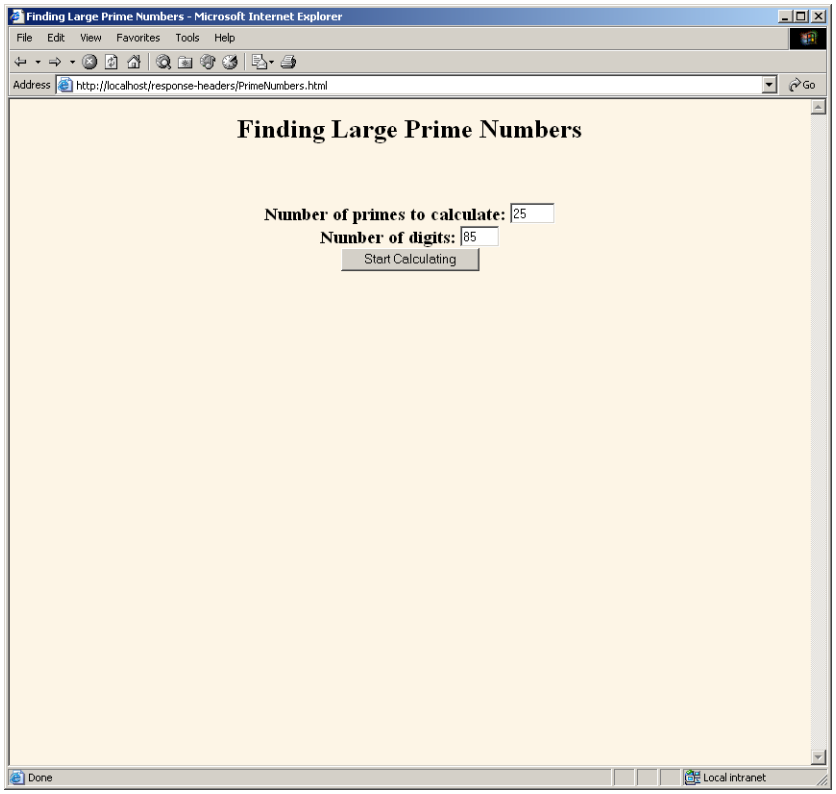


Figure 7–2 Front end to the prime-number-generation servlet.

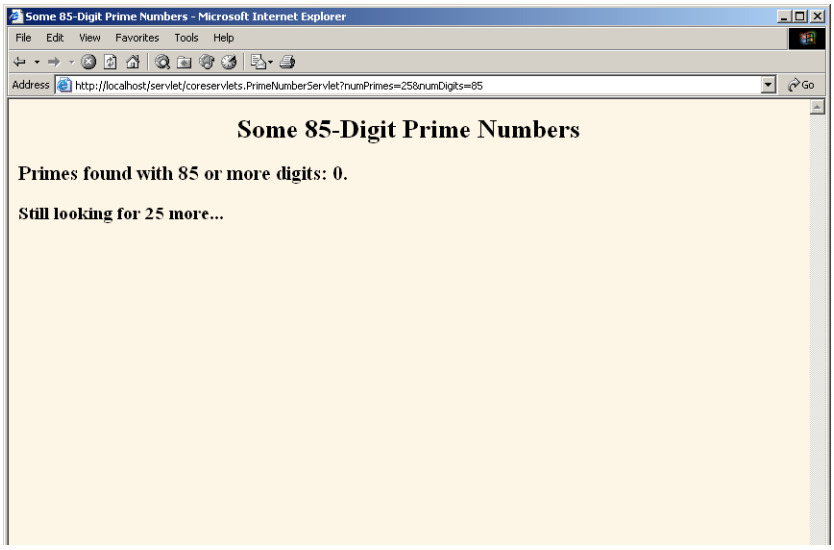


Figure 7–3 Initial results of the prime-number-generation servlet. A quick result is sent to the browser, along with instructions (in the `Refresh` header) to reconnect for an update in five seconds.

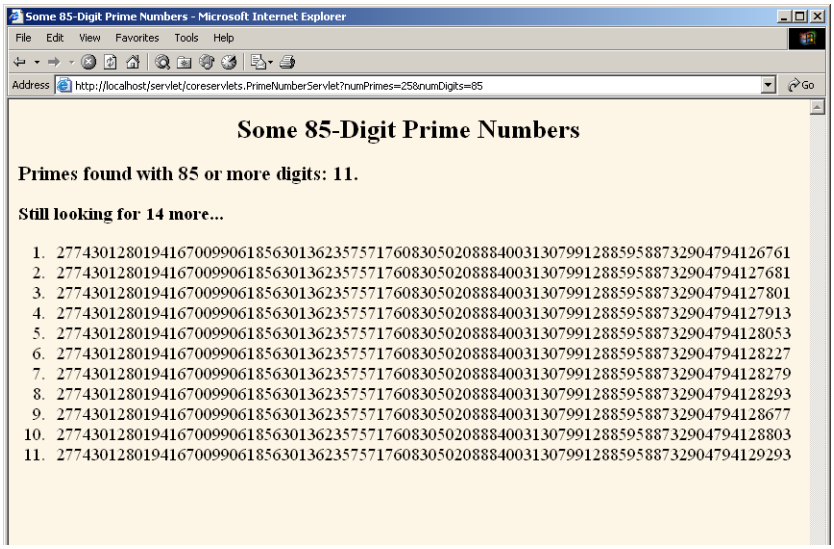


Figure 7–4 Intermediate results of the prime-number-generation servlet. The servlet stores the previous computations and matches the current request with the stored values by comparing the request parameters (the size and number of primes to compute). Other clients that request the same parameters see the same already computed results.

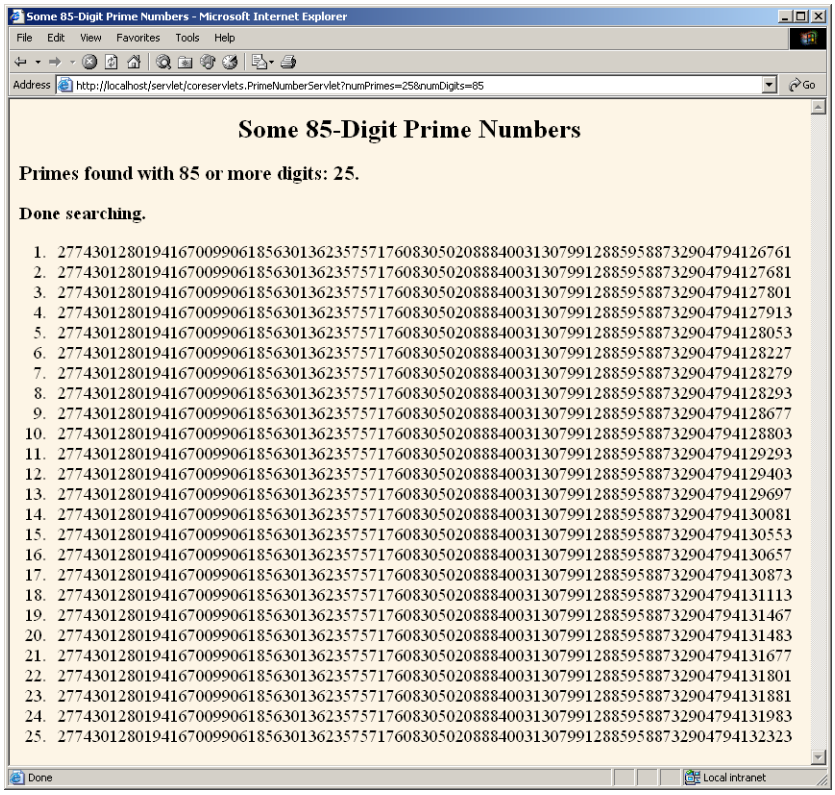


Figure 7–5 Final results of the prime-number-generation servlet. Since the servlet has computed as many primes as the user requested, no `Refresh` header is sent to the browser and the page is no longer reloaded automatically.

Listings 7.4 (PrimeList.java) and 7.5 (Primes.java) present auxiliary code used by the servlet. PrimeList.java handles the background thread for the creation of a list of primes for a specific set of values. The point of this example is twofold: that servlets can maintain data between requests by storing it in instance variables (or the `ServletContext`) and that the servlet can use the `Refresh` header to instruct the browser to return for updates. However, if you care about the gory details of prime-number generation, Primes.java contains the low-level algorithms for choosing a random number of a specified length and then finding a prime at or above that value. It uses built-in methods in the `BigInteger` class; the algorithm for determining if the number is prime is a probabilistic one and thus has a chance of being mistaken. However, the probability of being wrong can be specified, and we use an error value of 100. Assuming that the algorithm used in most Java implementations is the

Miller-Rabin test, the likelihood of falsely reporting a composite (i.e., non-prime) number as prime is provably less than 2^{100}. This is almost certainly smaller than the likelihood of a hardware error or random radiation causing an incorrect response in a deterministic algorithm, and thus the algorithm can be considered deterministic.

Listing 7.4 PrimeList.java

```
package coreservlets;

import java.util.*;
import java.math.BigInteger;

/** Creates an ArrayList of large prime numbers, usually in
 *  a low-priority background thread. Provides a few small
 *  thread-safe access methods.
 */

public class PrimeList implements Runnable {
  private ArrayList primesFound;
  private int numPrimes, numDigits;

  /** Finds numPrimes prime numbers, each of which is
   *  numDigits long or longer. You can set it to return
   *  only when done, or have it return immediately,
   *  and you can later poll it to see how far it
   *  has gotten.
   */

  public PrimeList(int numPrimes, int numDigits,
                   boolean runInBackground) {
    primesFound = new ArrayList(numPrimes);
    this.numPrimes = numPrimes;
    this.numDigits = numDigits;
    if (runInBackground) {
      Thread t = new Thread(this);
      // Use low priority so you don't slow down server.
      t.setPriority(Thread.MIN_PRIORITY);
      t.start();
    } else {
      run();
    }
  }
```

Listing 7.4 PrimeList.java *(continued)*

```
  public void run() {
    BigInteger start = Primes.random(numDigits);
    for(int i=0; i<numPrimes; i++) {
      start = Primes.nextPrime(start);
      synchronized(this) {
        primesFound.add(start);
      }
    }
  }

  public synchronized boolean isDone() {
    return(primesFound.size() == numPrimes);
  }

  public synchronized ArrayList getPrimes() {
    if (isDone())
      return(primesFound);
    else
      return((ArrayList)primesFound.clone());
  }

  public int numDigits() {
    return(numDigits);
  }

  public int numPrimes() {
    return(numPrimes);
  }

  public synchronized int numCalculatedPrimes() {
    return(primesFound.size());
  }
}
```

Listing 7.5 Primes.java

```
package coreservlets;

import java.math.BigInteger;

/** A few utilities to generate a large random BigInteger,
 *  and find the next prime number above a given BigInteger.
 */

public class Primes {
  // Note that BigInteger.ZERO and BigInteger.ONE are
  // unavailable in JDK 1.1.
  private static final BigInteger ZERO = BigInteger.ZERO;
  private static final BigInteger ONE = BigInteger.ONE;
  private static final BigInteger TWO = new BigInteger("2");

  // Likelihood of false prime is less than 1/2^ERR_VAL.
  // Presumably BigInteger uses the Miller-Rabin test or
  // equivalent, and thus is NOT fooled by Carmichael numbers.
  // See section 33.8 of Cormen et al.'s Introduction to
  // Algorithms for details.
  private static final int ERR_VAL = 100;

  public static BigInteger nextPrime(BigInteger start) {
    if (isEven(start))
      start = start.add(ONE);
    else
      start = start.add(TWO);
    if (start.isProbablePrime(ERR_VAL))
      return(start);
    else
      return(nextPrime(start));
  }

  private static boolean isEven(BigInteger n) {
    return(n.mod(TWO).equals(ZERO));
  }

  private static StringBuffer[] digits =
    { new StringBuffer("0"), new StringBuffer("1"),
      new StringBuffer("2"), new StringBuffer("3"),
      new StringBuffer("4"), new StringBuffer("5"),
      new StringBuffer("6"), new StringBuffer("7"),
      new StringBuffer("8"), new StringBuffer("9") };
```

Listing 7.5 Primes.java *(continued)*

```
  private static StringBuffer randomDigit(boolean isZeroOK) {
    int index;
    if (isZeroOK) {
      index = (int)Math.floor(Math.random() * 10);
    } else {
      index = 1 + (int)Math.floor(Math.random() * 9);
    }
    return(digits[index]);
  }

  /** Create a random big integer where every digit is
   *  selected randomly (except that the first digit
   *  cannot be a zero).
   */

  public static BigInteger random(int numDigits) {
    StringBuffer s = new StringBuffer("");
    for(int i=0; i<numDigits; i++) {
      if (i == 0) {
        // First digit must be non-zero.
        s.append(randomDigit(false));
      } else {
        s.append(randomDigit(true));
      }
    }
    return(new BigInteger(s.toString()));
  }

  /** Simple command-line program to test. Enter number
   *  of digits, and the program picks a random number of that
   *  length and then prints the first 50 prime numbers
   *  above that.
   */

  public static void main(String[] args) {
    int numDigits;
    try {
      numDigits = Integer.parseInt(args[0]);
    } catch (Exception e) { // No args or illegal arg.
      numDigits = 150;
    }
    BigInteger start = random(numDigits);
    for(int i=0; i<50; i++) {
      start = nextPrime(start);
      System.out.println("Prime " + i + " = " + start);
    }
  }
}
```

Listing 7.6 ServletUtilities.java (Excerpt)

```
package coreservlets;

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

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

public class ServletUtilities {
  // ...

  /** Read a parameter with the specified name, convert it
   *  to an int, and return it. Return the designated default
   *  value if the parameter doesn't exist or if it is an
   *  illegal integer format.
   */

  public static int getIntParameter(HttpServletRequest request,
                                    String paramName,
                                    int defaultValue) {
    String paramString = request.getParameter(paramName);
    int paramValue;
    try {
      paramValue = Integer.parseInt(paramString);
    } catch(NumberFormatException nfe) { // null or bad format
      paramValue = defaultValue;
    }
    return(paramValue);
  }
}
```

7.5 Using Servlets to Generate JPEG Images

Although servlets often generate HTML output, they certainly don't *always* do so. For example, Section 7.3 (Building Excel Spreadsheets) shows a servlet that builds Excel spreadsheets and returns them to the client. Here, we show you how to generate JPEG images.

First, let us summarize the two main steps servlets have to perform to build multimedia content.

1. **Inform the browser of the content type they are sending.** To accomplish this task, servlets set the `Content-Type` response header by using the `setContentType` method of `HttpServletResponse`.
2. **Send the output in the appropriate format.** This format varies among document types, of course, but in most cases you send binary data, not strings as you do with HTML documents. Consequently, servlets will usually get the raw output stream by using the `getOutputStream` method, rather than getting a `PrintWriter` by using `getWriter`.

Putting these two steps together, servlets that generate non-HTML content usually have a section of their `doGet` or `doPost` method that looks like this:

```
response.setContentType("type/subtype");
OutputStream out = response.getOutputStream();
```

Those are the two general steps required to build non-HTML content. Next, let's look at the specific steps required to generate JPEG images.

1. **Create a `BufferedImage`.**
   You create a `java.awt.image.BufferedImage` object by calling the `BufferedImage` constructor with a width, a height, and an image representation type as defined by one of the constants in the `BufferedImage` class. The representation type is not important, since we do not manipulate the bits of the `BufferedImage` directly and since most types yield identical results when converted to JPEG. We use `TYPE_INT_RGB`. Putting this all together, here is the normal process:

   ```
   int width = ...;
   int height = ...;
   BufferedImage image =
     new BufferedImage(width, height,
                       BufferedImage.TYPE_INT_RGB);
   ```

2. **Draw into the `BufferedImage`.**
   You accomplish this task by calling the image's `getGraphics` method, casting the resultant `Graphics` object to `Graphics2D`, then making use of Java 2D's rich set of drawing operations, coordinate transformations, font settings, and fill patterns to perform the drawing. Here is a simple example.

   ```
   Graphics2D g2d = (Graphics2D)image.getGraphics();
   g2d.setXxx(...);
   g2d.fill(someShape);
   g2d.draw(someShape);
   ```

3. **Set the `Content-Type` response header.**
   As already discussed, you use the `setContentType` method of `HttpServletResponse` for this task. The MIME type for JPEG images is `image/jpeg`. Thus, the code is as follows.

   ```
   response.setContentType("image/jpeg");
   ```

4. **Get an output stream.**
   As discussed previously, if you are sending binary data, you should call the `getOutputStream` method of `HttpServletResponse` rather than the `getWriter` method. For instance:

   ```
   OutputStream out = response.getOutputStream();
   ```

5. **Send the `BufferedImage` in JPEG format to the output stream.**
   Before JDK 1.4, accomplishing this task yourself required quite a bit of work. So, most people used a third-party utility for this purpose. In JDK 1.4 and later, however, the `ImageIO` class greatly simplifies this task. If you are using an application server that supports J2EE 1.4 (which includes servlets 2.4 and JSP 2.0), you are guaranteed to have JDK 1.4 or later. However, standalone servers are not absolutely required to use JDK 1.4, so be aware that this code depends on the Java version. When you use the `ImageIO` class, you just pass a `BufferedImage`, an image format type (`"jpg"`, `"png"`, etc.—call `ImageIO.getWriterFormatNames` for a complete list), and either an `OutputStream` or a `File` to the `write` method of `ImageIO`. Except for catching the required `IOException`, that's it! For example:

   ```
   try {
     ImageIO.write(image, "jpg", out);
   } catch(IOException ioe) {
     System.err.println("Error writing JPEG file: " + ioe);
   }
   ```

Listing 7.7 shows a servlet that reads `message`, `fontName`, and `fontSize` parameters and passes them to the `MessageImage` utility (Listing 7.8) to create a JPEG image showing the message in the designated face and size, with a gray, oblique-shadowed version of the message shown behind the main string. If the user presses the Show Font List button, then instead of building an image, the servlet displays a list of font names available on the server.

Listing 7.7 ShadowedText.java

```
package coreservlets;

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

/** Servlet that generates JPEG images representing
 *  a designated message with an oblique-shadowed
 *  version behind it.
 */

public class ShadowedText extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    String wantsList = request.getParameter("showList");
    if (wantsList != null) {
      showFontList(response);
    } else {
      String message = request.getParameter("message");
      if ((message == null) || (message.length() == 0)) {
        message = "Missing 'message' parameter";
      }
      String fontName = request.getParameter("fontName");
      if ((fontName == null) || (fontName.length() == 0)) {
        fontName = "Serif";
      }
      String fontSizeString = request.getParameter("fontSize");
      int fontSize;
      try {
        fontSize = Integer.parseInt(fontSizeString);
      } catch(NumberFormatException nfe) {
        fontSize = 90;
      }
      response.setContentType("image/jpeg");
      MessageImage.writeJPEG
        (MessageImage.makeMessageImage(message,
                                       fontName,
                                       fontSize),
         response.getOutputStream());
    }
  }
```

Listing 7.7 ShadowedText.java *(continued)*

```
  private void showFontList(HttpServletResponse response)
      throws IOException {
    PrintWriter out = response.getWriter();
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    String title = "Fonts Available on Server";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                "<UL>");
    String[] fontNames = MessageImage.getFontNames();
    for(int i=0; i<fontNames.length; i++) {
      out.println("  <LI>" + fontNames[i]);
    }
    out.println("</UL>\n" +
                "</BODY></HTML>");
  }
}
```

Listing 7.8 MessageImage.java

```
package coreservlets;

import java.awt.*;
import java.awt.geom.*;
import java.awt.image.*;
import java.io.*;
import javax.imageio.*;

/** Utilities for building images showing shadowed messages.
 *  <P>
 *  Requires JDK 1.4 since it uses the ImageIO class.
 *  JDK 1.4 is standard with J2EE-compliant app servers
 *  with servlets 2.4 and JSP 2.0. However, standalone
 *  servlet/JSP engines require only JDK 1.3 or later, and
 *  version 2.3 of the servlet spec requires only JDK
 *  1.2 or later. So, although most servers run on JDK 1.4,
 *  this code is not necessarily portable across all servers.
 */
```

Listing 7.8 MessageImage.java *(continued)*

```
public class MessageImage {

  /** Creates an Image of a string with an oblique
   *  shadow behind it. Used by the ShadowedText servlet.
   */

  public static BufferedImage makeMessageImage(String message,
                                               String fontName,
                                               int fontSize) {

    Font font = new Font(fontName, Font.PLAIN, fontSize);
    FontMetrics metrics = getFontMetrics(font);
    int messageWidth = metrics.stringWidth(message);
    int baselineX = messageWidth/10;
    int width = messageWidth+2*(baselineX + fontSize);
    int height = fontSize*7/2;
    int baselineY = height*8/10;
    BufferedImage messageImage =
      new BufferedImage(width, height,
                        BufferedImage.TYPE_INT_RGB);
    Graphics2D g2d = (Graphics2D)messageImage.getGraphics();
    g2d.setBackground(Color.white);
    g2d.clearRect(0, 0, width, height);
    g2d.setFont(font);
    g2d.translate(baselineX, baselineY);
    g2d.setPaint(Color.lightGray);
    AffineTransform origTransform = g2d.getTransform();
    g2d.shear(-0.95, 0);
    g2d.scale(1, 3);
    g2d.drawString(message, 0, 0);
    g2d.setTransform(origTransform);
    g2d.setPaint(Color.black);
    g2d.drawString(message, 0, 0);
    return(messageImage);
  }

  public static void writeJPEG(BufferedImage image,
                               OutputStream out) {
    try {
      ImageIO.write(image, "jpg", out);
    } catch(IOException ioe) {
      System.err.println("Error outputting JPEG: " + ioe);
    }
  }
```

Listing 7.8 MessageImage.java *(continued)*

```
  public static void writeJPEG(BufferedImage image,
                               File file) {
    try {
      ImageIO.write(image, "jpg", file);
    } catch(IOException ioe) {
      System.err.println("Error writing JPEG file: " + ioe);
    }
  }

  public static String[] getFontNames() {
    GraphicsEnvironment env =
      GraphicsEnvironment.getLocalGraphicsEnvironment();
    return(env.getAvailableFontFamilyNames());
  }

  /** We need a Graphics object to get a FontMetrics object
   *  (an object that says how big strings are in given fonts).
   *  But, you need an image from which to derive the Graphics
   *  object. Since the size of the "real" image will depend on
   *  how big the string is, we create a very small temporary
   *  image first, get the FontMetrics, figure out how
   *  big the real image should be, then use a real image
   *  of that size.
   */

  private static FontMetrics getFontMetrics(Font font) {
    BufferedImage tempImage =
      new BufferedImage(1, 1, BufferedImage.TYPE_INT_RGB);
    Graphics2D g2d = (Graphics2D)tempImage.getGraphics();
    return(g2d.getFontMetrics(font));
  }
}
```

Listing 7.9 (Figure 7–6) shows an HTML form used as a front end to the servlet. Figures 7–7 through 7–10 show some possible results. Just to simplify experimentation, Listing 7.10 presents an interactive application that lets you specify the message and font name on the command line, outputting the image to a file.

Listing 7.9 ShadowedText.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>JPEG Generation Service</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN="CENTER">JPEG Generation Service</H1>
Welcome to the <I>free</I> trial edition of our JPEG
generation service. Enter a message, a font name,
and a font size below, then submit the form. You will
be returned a JPEG image showing the message in the
designated font, with an oblique "shadow" of the message
behind it. Once you get an image you are satisfied with,
right-click
on it (or click while holding down the SHIFT key) to save
it to your local disk.
<P>
The server is currently on Windows, so the font name must
be either a standard Java font name (e.g., Serif, SansSerif,
or Monospaced) or a Windows font name (e.g., Arial Black).
Unrecognized font names will revert to Serif. Press the
"Show Font List" button for a complete list.

<FORM ACTION="/servlet/coreservlets.ShadowedText">
  <CENTER>
    Message:
    <INPUT TYPE="TEXT" NAME="message"><BR>
    Font name:
    <INPUT TYPE="TEXT" NAME="fontName" VALUE="Serif"><BR>
    Font size:
    <INPUT TYPE="TEXT" NAME="fontSize" VALUE="90"><P>
    <INPUT TYPE="SUBMIT" VALUE="Build Image"><P>
    <INPUT TYPE="SUBMIT" NAME="showList" VALUE="Show Font List">
  </CENTER>
</FORM>

</BODY></HTML>
```

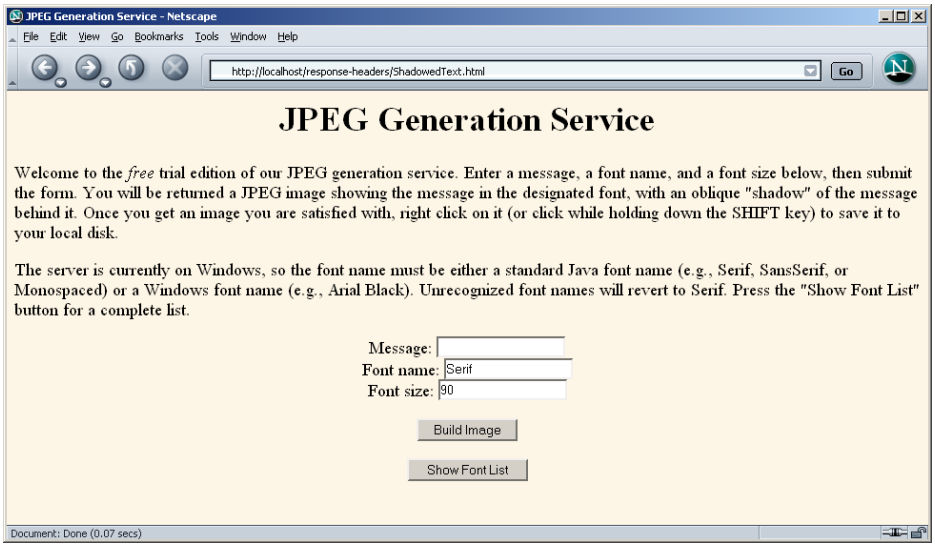


Figure 7–6 Front end to the image-generation servlet.

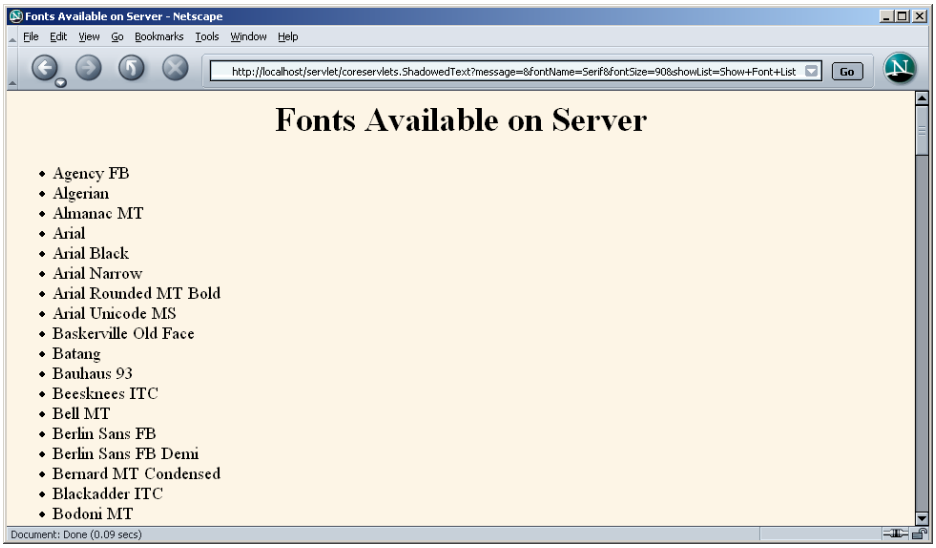


Figure 7–7 Result of servlet when the client selects Show Font List.

Figure 7–8 One possible result of the image-generation servlet. The client can save the image to disk as *somename*.jpg and use it in Web pages or other applications.

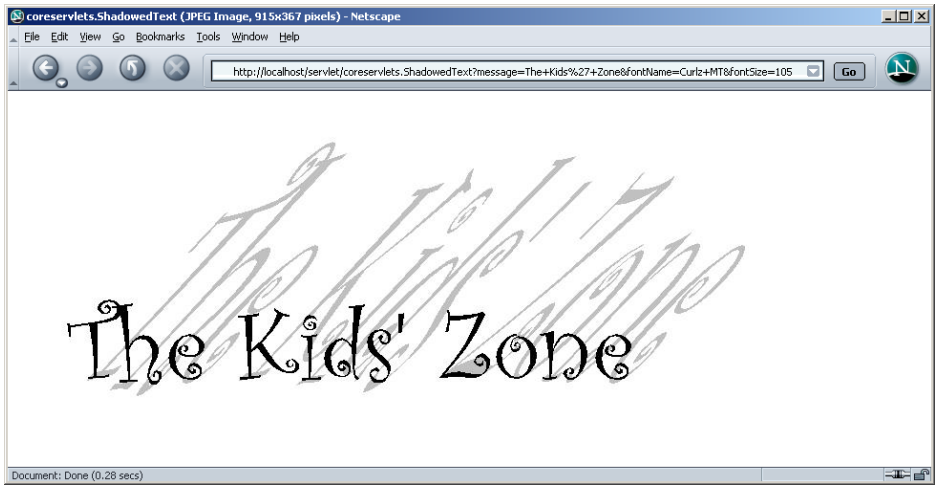


Figure 7–9 A second possible result of the image-generation servlet.

Figure 7–10 A third possible result of the image-generation servlet.

Listing 7.10 ImageTest.java

```
package coreservlets;

import java.io.*;

public class ImageTest {
  public static void main(String[] args) {
    String message = "Testing";
    String font = "Arial";
    if (args.length > 0) {
      message = args[0];
    }
    if (args.length > 1) {
      font = args[1];
    }
    MessageImage.writeJPEG
      (MessageImage.makeMessageImage(message, font, 40),
       new File("ImageTest.jpg"));
  }
}
```