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)");
  }
}
```

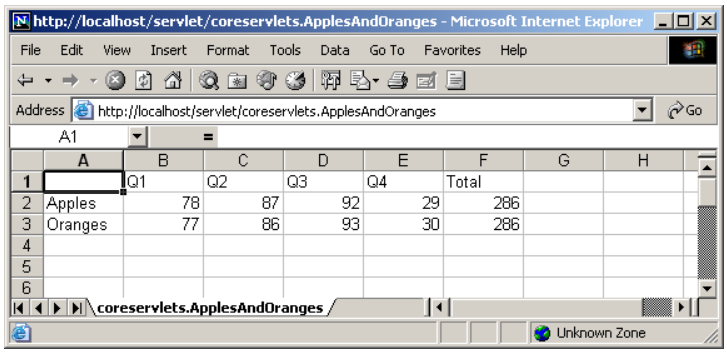

Figure 7–1 Result of the `ApplesAndOranges` servlet in Internet Explorer on a system that has Microsoft Office installed.

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>
```

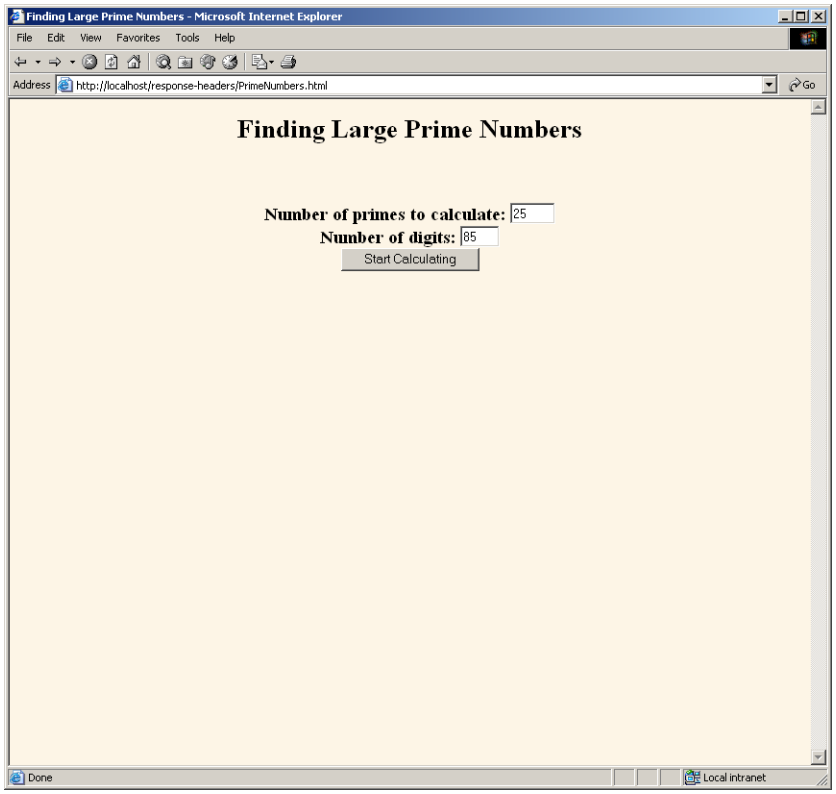

Figure 7–2 Front end to the prime-number-generation servlet.

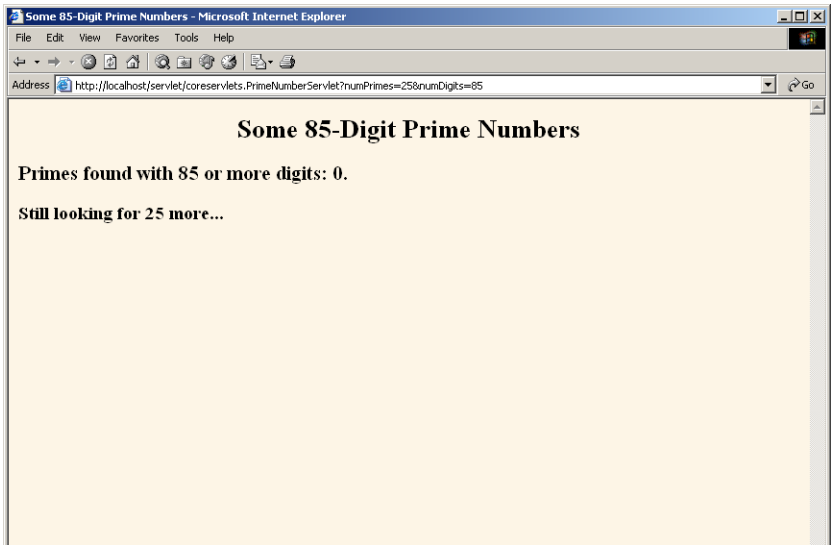

Figure 7–3 Initial results of the prime-number-generation servlet. A quick result is sent to the browser, along with instructions (in the `Refresh` header) to reconnect for an update in five seconds.

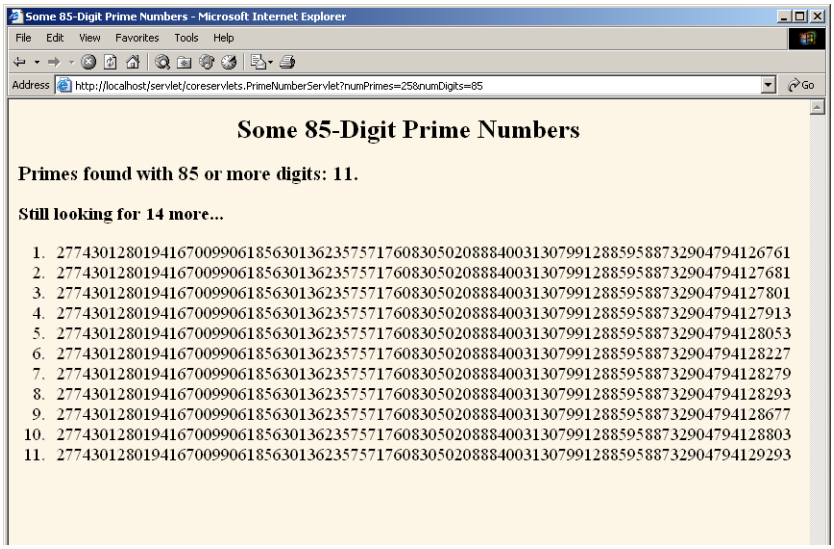

Figure 7–4 Intermediate results of the prime-number-generation servlet. The servlet stores the previous computations and matches the current request with the stored values by comparing the request parameters (the size and number of primes to compute). Other clients that request the same parameters see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

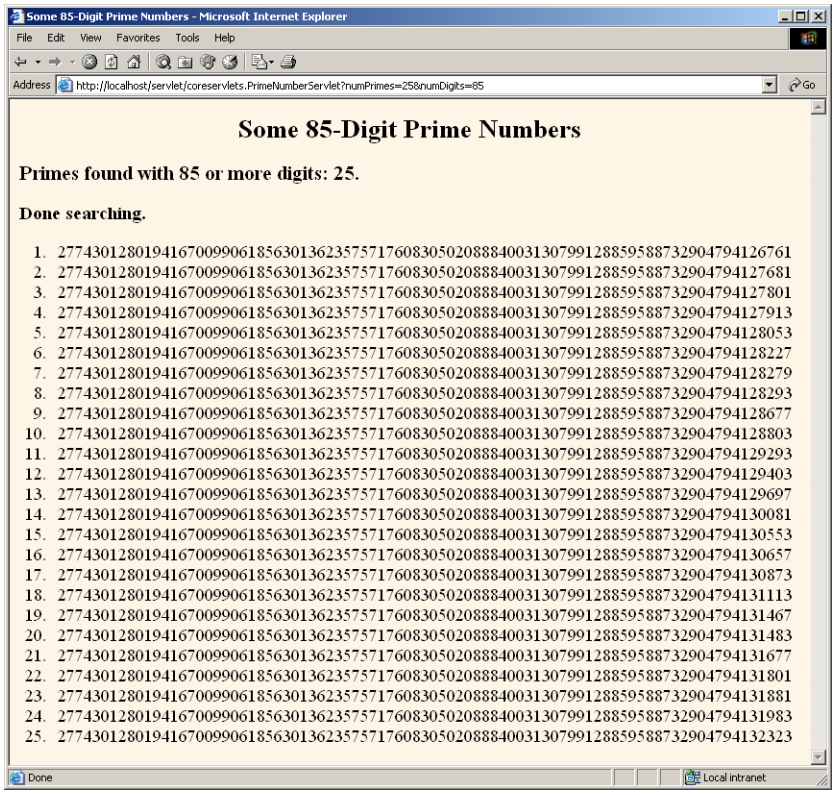

Figure 7–5 Final results of the prime-number-generation servlet. Since the servlet has computed as many primes as the user requested, no `Refresh` header is sent to the browser and the page is no longer reloaded automatically.

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>
```

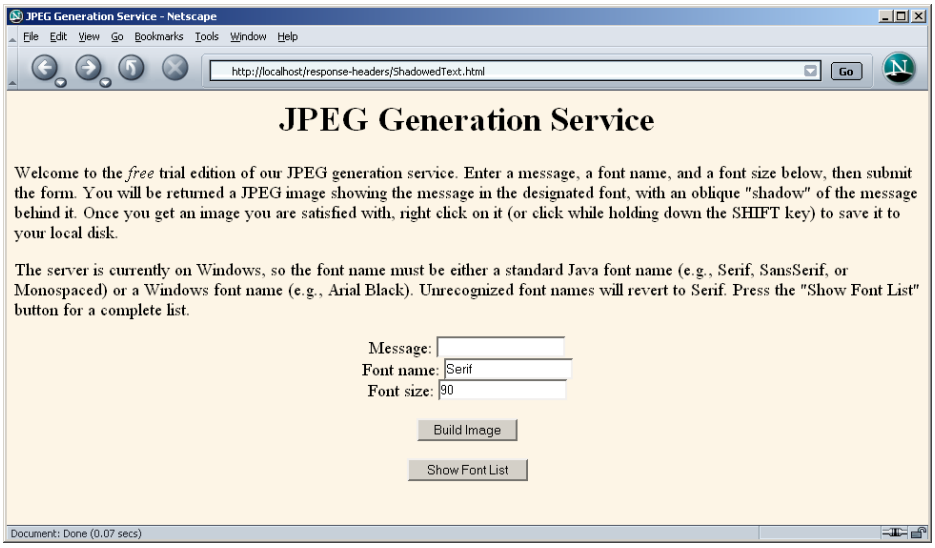

Figure 7–6 Front end to the image-generation servlet.

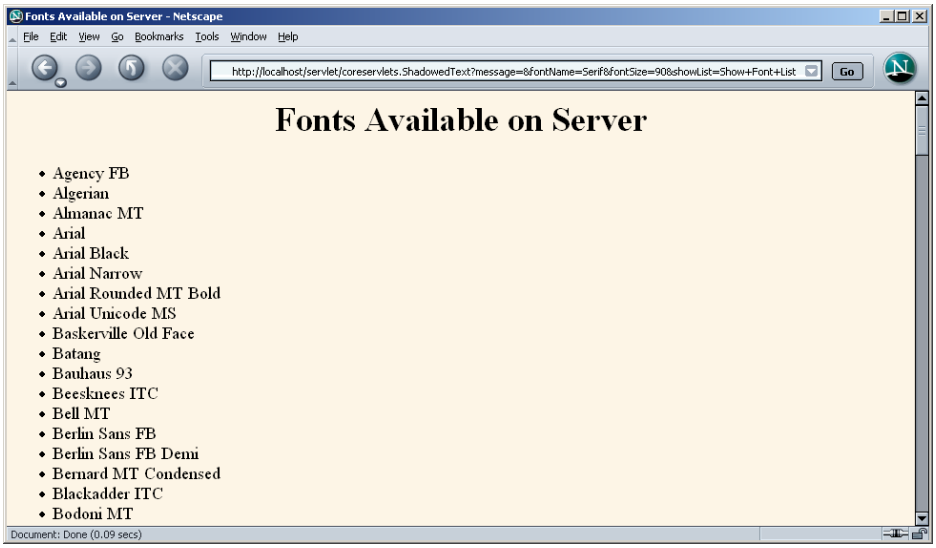

Figure 7–7 Result of servlet when the client selects Show Font List.

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.

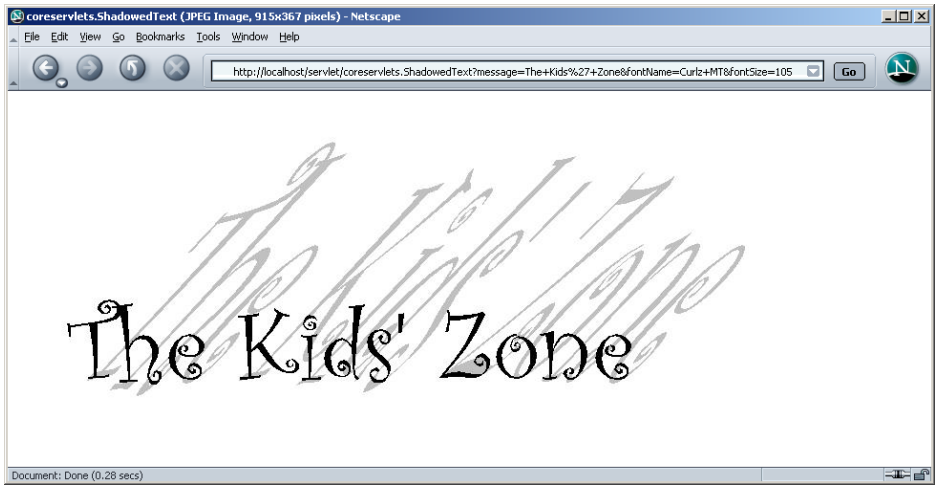

Figure 7–9 A second possible result of the image-generation servlet.

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)");
  }
}
```

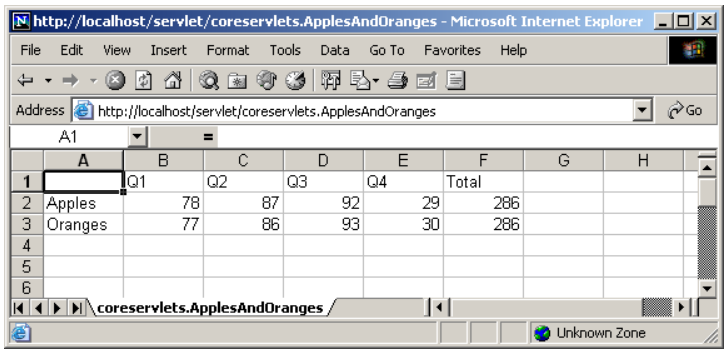

Figure 7–1 Result of the `ApplesAndOranges` servlet in Internet Explorer on a system that has Microsoft Office installed.

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>
```

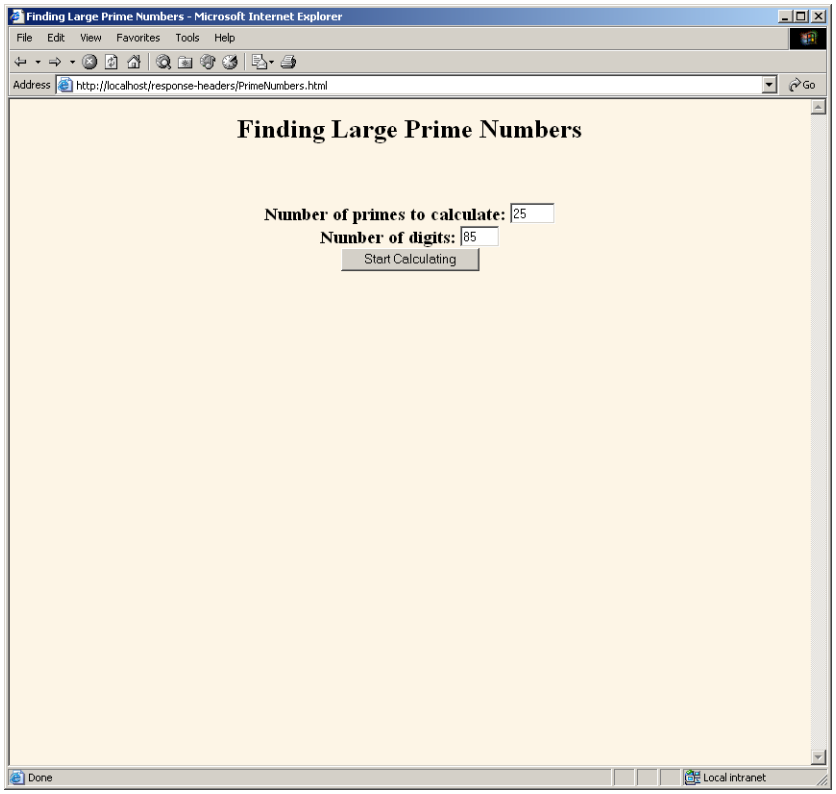

Figure 7–2 Front end to the prime-number-generation servlet.

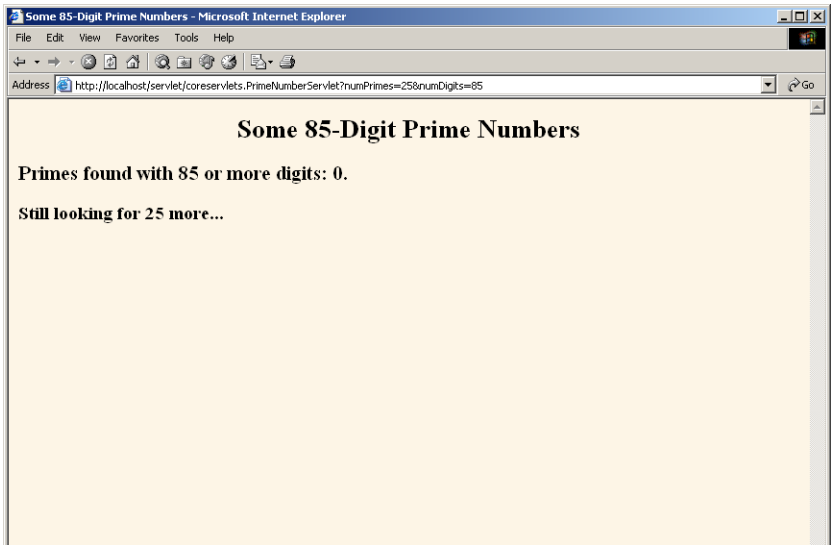

Figure 7–3 Initial results of the prime-number-generation servlet. A quick result is sent to the browser, along with instructions (in the `Refresh` header) to reconnect for an update in five seconds.

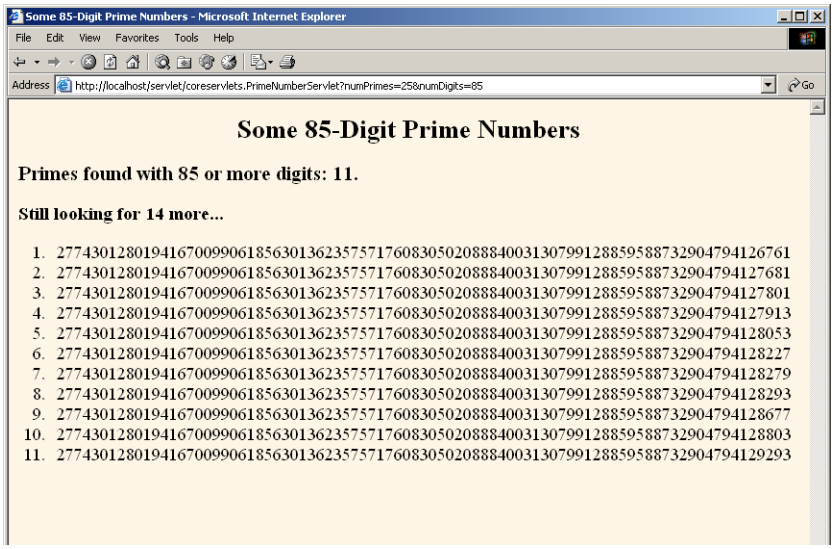

Figure 7–4 Intermediate results of the prime-number-generation servlet. The servlet stores the previous computations and matches the current request with the stored values by comparing the request parameters (the size and number of primes to compute). Other clients that request the same parameters see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

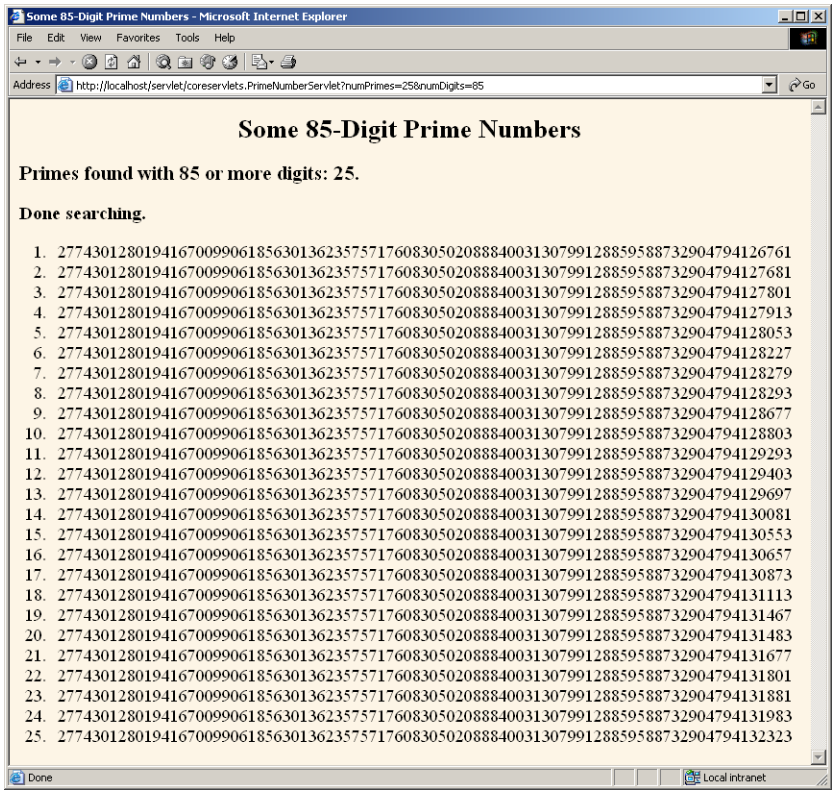

Figure 7–5 Final results of the prime-number-generation servlet. Since the servlet has computed as many primes as the user requested, no `Refresh` header is sent to the browser and the page is no longer reloaded automatically.

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>
```

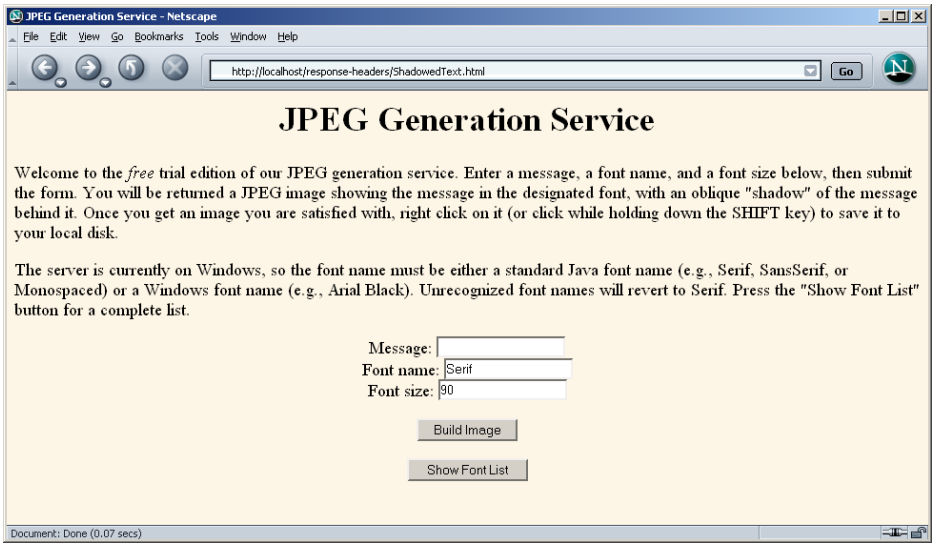

Figure 7–6 Front end to the image-generation servlet.

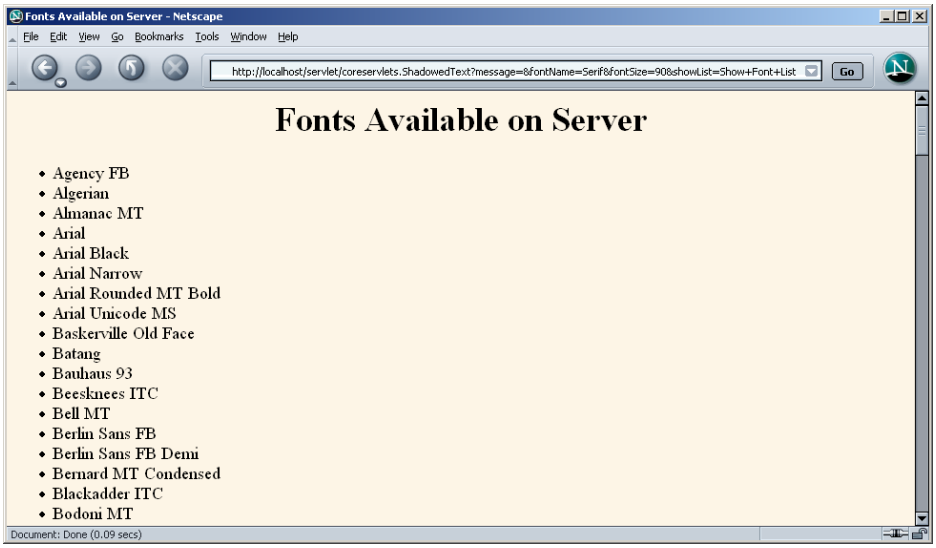

Figure 7–7 Result of servlet when the client selects Show Font List.

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.

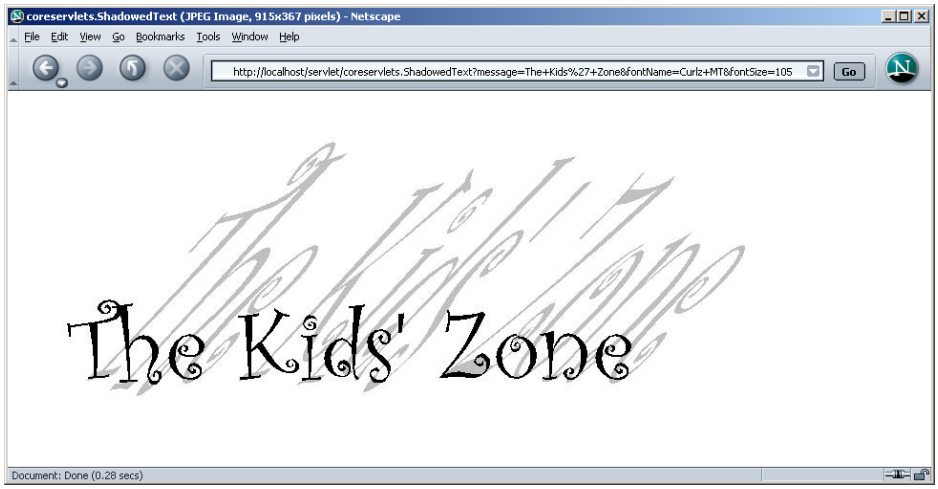

Figure 7–9 A second possible result of the image-generation servlet.

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"));
  }
}
```