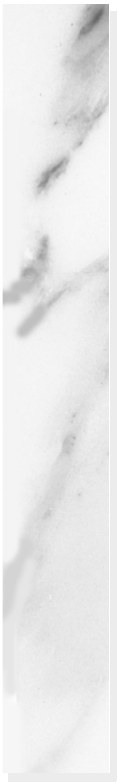
GENERATING THE SERVER RESPONSE: HTTP RESPONSE HEADERS



Topics in This Chapter

- Setting response headers from servlets
- The purpose of each of the HTTP 1.1 response headers
- Common MIME types
- A servlet that uses the `Refresh` header to repeatedly access ongoing computations
- Servlets that exploit persistent (keep-alive) HTTP connections
- Generating GIF images from servlets

Chapter 7

A response from a Web server normally consists of a status line, one or more response headers, 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.

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. With servlets version 2.1, this means that you must set the headers before the first use of the `PrintWriter` or raw `OutputStream` that transmits the document content. With servlets version 2.2 (the version in J2EE), the `PrintWriter` may use a buffer, so you can set headers until the first time the buffer is flushed. See Section 6.1 (Specifying Status Codes) for details.

Core Approach

*Be sure to set response headers **before** sending any document content to the client.*

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

- **`setDateHeader(String header, long milliseconds)`**
  This method saves you the trouble of translating a Java date in milliseconds since 1970 (as returned by `System.currentTimeMillis`, `Date.getTime`, or `Calendar.getTimeInMillis`) into a GMT time string.
- **`setIntHeader(String header, int headerValue)`**
  This method spares you the minor inconvenience of converting an `int` to a `String` before inserting it into a header.

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

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

- **`setContentType`**
  This method sets the `Content-Type` header and is used by the majority of servlets. See Section 7.5 (Using Servlets to Generate GIF Images) for an example of its use.
- **`setContentLength`**
  This method sets the `Content-Length` header, which is useful if the browser supports persistent (keep-alive) HTTP connections. See Section 7.4 for an example.
- **`addCookie`**
  This method inserts a cookie into the `Set-Cookie` header. There is no corresponding `setCookie` method, since it is normal to have multiple `Set-Cookie` lines. See Chapter 8 for a discussion of cookies.
- **`sendRedirect`**
  As discussed in the previous chapter, the `sendRedirect` method sets the `Location` header as well as setting the status code to 302. See Section 6.3 (A Front End to Various Search Engines) for an example.

7.2 HTTP 1.1 Response Headers and Their Meaning

Following is a summary of the 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 to get details when you are ready to make use of the capabilities. Note that Appendix A (Servlet and JSP Quick Reference) presents a brief summary of these headers for use as a reminder.

These headers are a superset of those permitted in HTTP 1.0. For additional details on these headers, see the HTTP 1.1 specification, given in RFC 2616. There are a number of places the official RFCs are archived on-line; 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 only available in HTTP 1.1, especially if your servlet needs to run on the WWW "at large," rather than on an intranet—many older browsers support only HTTP 1.0. It is best to explicitly check the HTTP version with `request.getRequestProtocol` before using new headers.

Accept-Ranges

This header, which is new in HTTP 1.1, tells the client whether or not you accept `Range` request headers. You typically specify a value of `bytes` to indicate that you accept `Range` requests, and a value of `none` to indicate that you do not.

Age

This header is used by proxies to indicate how long ago the document was generated by the original server. It is new in HTTP 1.1 and is rarely used by servlets.

Allow

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

Cache-Control

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

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

```
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. See Section 7.4 (Using Persistent HTTP Connections) for an example of the use of persistent HTTP connections from servlets.

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 4.4 (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 (you can access RFCs on-line at one of the archive sites listed at `http://www.rfc-editor.org/`).

Content-Length

This header indicates the number of bytes in the response. This information is needed only if the browser is using a persistent (keep-alive) HTTP connection. See the `Connection` header for determining when the browser supports persistent connections. If you want your servlet to take advantage of persistent connections when the browser supports it, your servlet should write the document into a `ByteArrayOutputStream`, look up its size when done, put that into the `Content-Length` field with `response.setContentLength`, then send the content via `byteArrayStream.writeTo(response.getOutputStream())`. For an example of this approach, see Section 7.4.

Content-Location

This header supplies an alternative address for the requested document. `Content-Location` is informational; responses that include this header also include the requested document, unlike the case with the `Location` header. This header is new to HTTP 1.1.

Content-MD5

The `Content-MD5` response header provides an MD5 digest for the subsequent document. This digest provides a message integrity check for clients that want to confirm they received the complete, unaltered document. See RFC 1864 for details on MD5. This header is new in HTTP 1.1.

Content-Range

This new HTTP 1.1 header is sent with partial-document responses and specifies how much of the total document was sent. For example, a value of "`bytes 500-999/2345`" means that the current response includes bytes 500 through 999 of a document that contains 2345 bytes in total.

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. The default MIME type for servlets is `text/plain`, but servlets usually explicitly specify `text/html`. They can, however, specify other types instead. For example, Section 7.5 (Using Servlets to Generate GIF Images) presents a servlet that builds a GIF image based upon input provided by specifying the `image/gif` content type, and Section 11.2 (The contentType Attribute) shows how servlets and JSP pages can generate Excel spreadsheets by specifying a content type of `application/vnd.ms-excel`.

Table 7.1 lists some the most common MIME types used by servlets.

For more detail, many of the common MIME types are listed in RFC 1521 and RFC 1522 (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
`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

Table 7.1 Common MIME Types (continued)	
`application/zip`	Zip archive
`audio/basic`	Sound file in .au or .snd format
`audio/x-aiff`	AIFF sound file
`audio/x-wav`	Microsoft Windows sound file
`audio/midi`	MIDI sound file
`text/css`	HTML cascading style sheet
`text/html`	HTML document
`text/plain`	Plain text
`image/gif`	GIF image
`image/jpeg`	JPEG image
`image/png`	PNG image
`image/tiff`	TIFF image
`image/x-xbitmap`	X Window bitmap image
`video/mpeg`	MPEG video clip
`video/quicktime`	QuickTime video clip

Date
This header specifies the current date in GMT format. If you want to set the date from a servlet, use the `setDateHeader` method to specify it. That method saves you the trouble of formatting the date string properly, as would be necessary with `response.setHeader("Date", "...")`. However, most servers set this header automatically, so servlets don't usually need to.

ETag
This new HTTP 1.1 header gives names to returned documents so that they can be referred to by the client later (as with the `If-Match` request header).

Expires
This header stipulates the time at which the content should be considered out-of-date and thus no longer be cached. A servlet might use this

for a document that changes relatively frequently, to prevent the browser from displaying a stale cached value. For example, the following would instruct the browser not to cache the document for longer 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 only being returned 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 and let the standard `service` method handle `If-Modified-Since` requests. For an example, see Section 2.8 (An Example Using Servlet Initialization and Page Modification Dates).

Location

This header, which should be included with all responses that have a status code in the 300s, notifies the browser of the document address. The browser automatically reconnects to this location and retrieves the new document. This header is usually set indirectly, along with a 302 status code, by the `sendRedirect` method of `HttpServletResponse`. An example is given in Section 6.3 (A Front End to Various Search Engines).

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. 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, and sending a 204 (`No Content`) status code stops the browser from reloading further. For an example, see Section 7.3 (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," where an introductory image or message is displayed briefly before the real page is loaded.

Note that this header is commonly set by

```
<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.

Server

This header identifies the Web server. Servlets don't usually set this; the Web server itself does.

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 very widely supported, including in both Netscape and Internet Explorer.

Trailer
This new and rarely used HTTP 1.1 header identifies the header fields that are present in the trailer of a message that is sent with "chunked" transfer-coding. See Section 3.6 of the HTTP 1.1 specification (RFC 2616) for details. Recall that `http://www.rfc-editor.org/` maintains an up-to-date list of RFC archive sites.

Transfer-Encoding
Supplying this header with a value of `chunked` indicates "chunked" transfer-coding. See Section 3.6 of the HTTP 1.1 specification (RFC 2616) for details.

Upgrade
This header is used when the client first uses the `Upgrade` *request* header to ask the server to switch to one of several possible new protocols. If the server agrees, it sends a 101 (`Switching Protocols`) status code and includes an `Upgrade` *response* header with the specific protocol it is switching to. This protocol negotiation is usually carried on by the server itself, not by a servlet.

Vary
This rarely used new HTTP 1.1 header tells the client which headers can be used to determine if the response document can be cached.

Via
This header is used by gateways and proxies to list the intermediate sites the request passed through. It is new in HTTP 1.1.

Warning
This new and rarely used catchall header lets you warn clients about caching or content transformation errors.

WWW-Authenticate
This header is always included with a 401 (`Unauthorized`) status code. It tells the browser what authorization type and realm the client should supply in its `Authorization` header. Frequently, servlets let password-protected Web pages be handled by the Web server's specialized mechanisms (e.g., `.htaccess`) rather than handling them directly. For an example of servlets dealing directly with this header, see Section 4.5 (Restricting Access to Web Pages).

7.3 Persistent Servlet State and Auto-Reloading Pages

Here is an example that lets you ask for a list of some large, randomly chosen prime numbers. This computation may take some time for very large numbers (e.g., 150 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, 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 traditional CGI and many CGI 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 in the servlet engine (or in the Web application, if your server supports such a capability).

Listing 7.1 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 7.2 shows the source code and Figure 7–1 shows the result. Next, these parameters are converted to integers by means of a simple utility that uses `Integer.parseInt` (see Listing 7.5). These values are then matched by the `findPrimeList` method to a `Vector` of recent or ongoing calculations to see if there is a previous computation corresponding 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 if 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.

Listings 7.3 (`PrimeList.java`) and 7.4 (`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. `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 I 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 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 7.1 PrimeNumbers.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 recent completed computation.
 */

public class PrimeNumbers extends HttpServlet {
  private Vector primeListVector = new Vector();
  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(primeListVector, 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(primeListVector) {
        if (primeListVector.size() >= maxPrimeLists)
          primeListVector.removeElementAt(0);
        primeListVector.addElement(primeList);
      }
    }
    Vector currentPrimes = primeList.getPrimes();
    int numCurrentPrimes = currentPrimes.size();
    int numPrimesRemaining = (numPrimes - numCurrentPrimes);
    boolean isLastResult = (numPrimesRemaining == 0);
    if (!isLastResult) {
      response.setHeader("Refresh", "5");
    }
```

Listing 7.1 PrimeNumbers.java (continued)

```
    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.elementAt(i));
    }
    out.println("</OL>");
    out.println("</BODY></HTML>");
  }

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

  // 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(Vector primeListVector,
                                  int numPrimes,
                                  int numDigits) {
    synchronized(primeListVector) {
      for(int i=0; i<primeListVector.size(); i++) {
        PrimeList primes =
          (PrimeList)primeListVector.elementAt(i);
        if ((numPrimes == primes.numPrimes()) &&
            (numDigits == primes.numDigits()))
          return(primes);
      }
      return(null);
    }
  }
}
```

Listing 7.2 PrimeNumbers.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Finding Large Prime Numbers</TITLE>
</HEAD>

<BODY BGCOLOR="#FDF5E6">
<H2 ALIGN="CENTER">Finding Large Prime Numbers</H2>
<BR><BR>
<CENTER>
<FORM ACTION="/servlet/coreservlets.PrimeNumbers">
  <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>
```

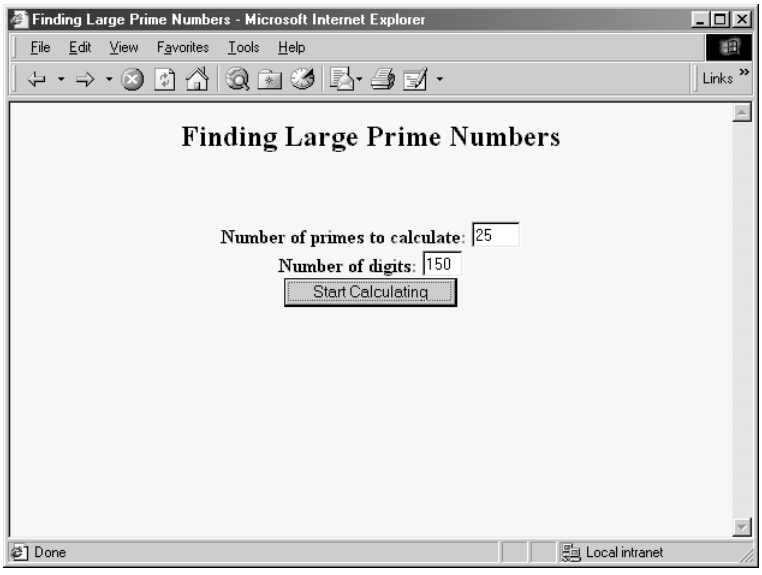


Figure 7–1 Result of `PrimeNumbers.html`, used as a front end to the `PrimeNumbers` servlet.

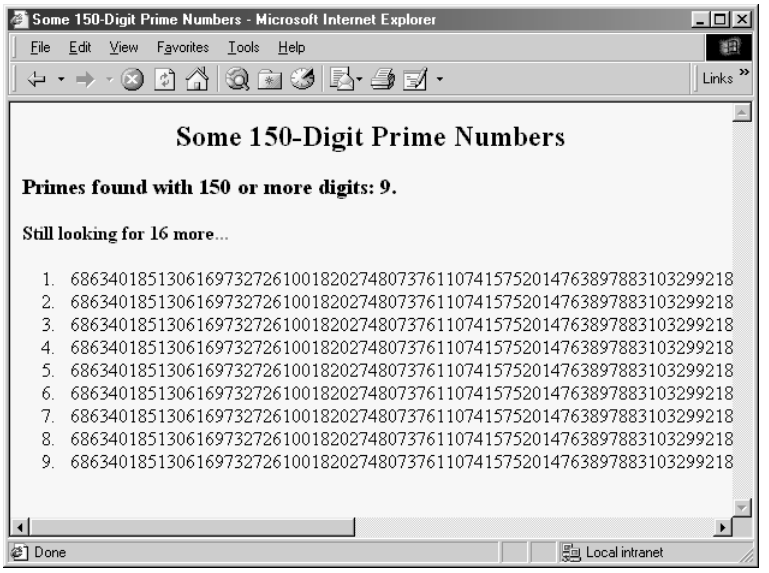


Figure 7–2 Intermediate result of a request to the `PrimeNumbers` servlet. This result can be obtained when the browser reloads automatically or when a different client independently enters the same parameters as those from an ongoing or recent request. Either way, the browser will automatically reload the page to get updated results.

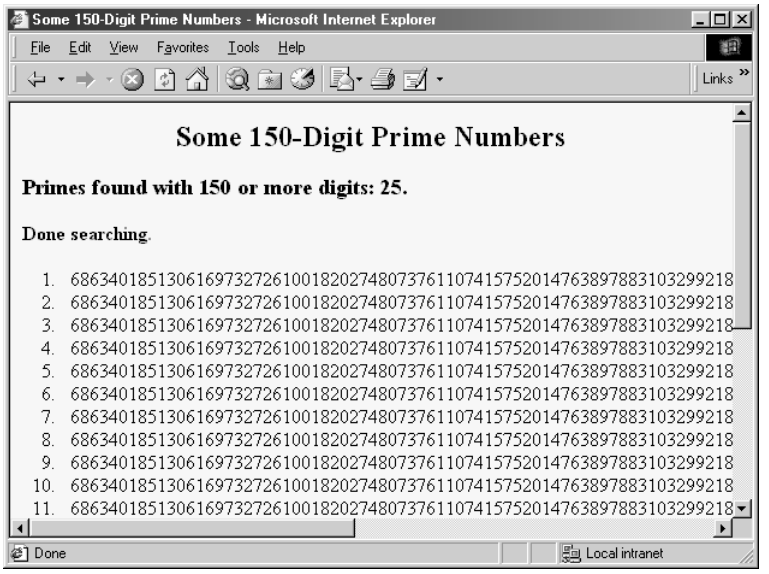


Figure 7–3 Final result of a request to the `PrimeNumbers` servlet. This result can be obtained when the browser reloads automatically or when a different client independently enters the same parameters as those from an ongoing or recent request. The browser will stop updating the page at this point.

Listing 7.3 PrimeList.java

```
package coreservlets;

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

/** Creates a Vector 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 Vector primesFound;
  private int numPrimes, numDigits;

  /** Finds numPrimes prime numbers, each of which are
   *  numDigits long or longer. You can set it to only
   *  return 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) {
    // Using Vector instead of ArrayList
    // to support JDK 1.1 servlet engines
    primesFound = new Vector(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();
    }
  }

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

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

Listing 7.3 PrimeList.java (continued)

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

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

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

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

Listing 7.4 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 was new in JDK 1.2, and 1.1
  // code is being used to support the most servlet engines.
  private static final BigInteger ZERO = new BigInteger("0");
  private static final BigInteger ONE = new BigInteger("1");
  private static final BigInteger TWO = new BigInteger("2");

  // Likelihood of false prime is less than 1/2^ERR_VAL
  // Assumedly 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
```

Listing 7.4 Primes.java (continued)

```
      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") };

  private static StringBuffer randomDigit() {
    int index = (int)Math.floor(Math.random() * 10);
    return(digits[index]);
  }

  public static BigInteger random(int numDigits) {
    StringBuffer s = new StringBuffer("");
    for(int i=0; i<numDigits; i++) {
      s.append(randomDigit());
    }
    return(new BigInteger(s.toString()));
  }

  /** Simple command-line program to test. Enter number
   *  of digits, and it 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;
    if (args.length > 0)
      numDigits = Integer.parseInt(args[0]);
    else
      numDigits = 150;
    BigInteger start = random(numDigits);
    for(int i=0; i<50; i++) {
      start = nextPrime(start);
      System.out.println("Prime " + i + " = " + start);
    }
  }
}
```

Listing 7.5 ServletUtilities.java

```
package coreservlets;

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


public class ServletUtilities {
  // ... Other utilities shown earlier

  /** 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);
  }

  // ...
}
```

7.4 Using Persistent HTTP Connections

One of the problems with HTTP 1.0 was that it required a separate socket connection for each request. When a Web page that includes lots of small images or many applet classes is retrieved, the overhead of establishing all the connections could be significant compared to the actual download time of the documents. Many browsers and servers supported the "keep-alive" extension to address this problem. With this extension, the server tells the browser how many bytes are contained in the response, then leaves the connection open for a certain period of time after returning the document. The client detects

that the document has finished loading by monitoring the number of bytes received, and reconnects on the same socket for further transactions. Persistent connections of this type became standard in HTTP 1.1, and compliant servers are supposed to use persistent connections unless the client explicitly instructs them not to (either by a "`Connection: close`" request header or indirectly by sending a request that specifies `HTTP/1.0` instead of `HTTP/1.1` and does *not* also stipulate "`Connection: keep-alive`").

Servlets can take advantage of persistent connections if the servlets are embedded in servers that support them. The server should handle most of the process, but it has no way to determine how large the returned document is. So the servlet needs to set the `Content-Length` response header by means of `response.setContentLength`. A servlet can determine the size of the returned document by buffering the output by means of a `ByteArrayOutputStream`, retrieving the number of bytes with the byte stream's `size` method, then sending the buffered output to the client by passing the servlet's output stream to the byte stream's `writeTo` method.

Using persistent connections is likely to pay off only for servlets that load a large number of small objects, where those objects are also servlet-generated and would thus not otherwise take advantage of the server's support for persistent connections. Even so, the advantage gained varies greatly from Web server to Web server and even from Web browser to Web browser. For example, the default configuration for Sun's Java Web Server is to permit only five connections on a single HTTP socket: a value that is too low for many applications. Those who use this server can raise the limit by going to the administration console, selecting "Web Service" then "Service Tuning," then entering a value in the "Connection Persistence" window.

Listing 7.6 shows a servlet that generates a page with 100 `IMG` tags (see Figure 7–4 for the result). Each of the `IMG` tags refers to another servlet (`ImageRetriever`, shown in Listing 7.7) that reads a GIF file from the server system and returns it to the client. Both the original servlet and the `ImageRetriever` servlet use persistent connections unless instructed not to do so by means of a parameter in the form data named `usePersistence` with a value of `no`. With Netscape 4.7 and a 28.8K dialup connection to talk to the Solaris version of Java Web Server 2.0 (with the connection limit raised above 100), the use of persistent connections reduced the average download time between 15 and 20 percent.

Listing 7.6 PersistentConnection.java

```
package coreservlets;

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

/** Illustrates the value of persistent HTTP connections for
 *  pages that include many images, applet classes, or
 *  other auxiliary content that would otherwise require
 *  a separate connection to retrieve.
 */

public class PersistentConnection extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    ByteArrayOutputStream byteStream =
      new ByteArrayOutputStream(7000);
    PrintWriter out = new PrintWriter(byteStream, true);
    String persistenceFlag =
      request.getParameter("usePersistence");
    boolean usePersistence =
      ((persistenceFlag == null) ||
       (!persistenceFlag.equals("no")));
    String title;
    if (usePersistence) {
      title = "Using Persistent Connection";
    } else {
      title = "Not Using Persistent Connection";
    }
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=\"CENTER\">" + title + "</H1>");
    int numImages = 100;
    for(int i=0; i<numImages; i++) {
      out.println(makeImage(i, usePersistence));
    }
    out.println("</BODY></HTML>");
    if (usePersistence) {
      response.setContentLength(byteStream.size());
    }
    byteStream.writeTo(response.getOutputStream());
  }

  private String makeImage(int n, boolean usePersistence) {
    String file =
      "/servlet/coreservlets.ImageRetriever?gifLocation=" +
      "/bullets/bullet" + n + ".gif";
    if (!usePersistence)
```

Listing 7.6 PersistentConnection.java (continued)

```
      file = file + "&usePersistence=no";
    return("<IMG SRC=\"" + file + "\"\n" +
           "     WIDTH=6 HEIGHT=6 ALT=\"\">");
  }

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

Listing 7.7 ImageRetriever.java

```
package coreservlets;

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

/** A servlet that reads a GIF file off the local system
 *  and sends it to the client with the appropriate MIME type.
 *  Includes the Content-Length header to support the
 *  use of persistent HTTP connections unless explicitly
 *  instructed not to through "usePersistence=no".
 *  Used by the PersistentConnection servlet.
*/

public class ImageRetriever extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    String gifLocation = request.getParameter("gifLocation");
    if ((gifLocation == null) ||
        (gifLocation.length() == 0)) {
      reportError(response, "Image File Not Specified");
      return;
    }
    String file = getServletContext().getRealPath(gifLocation);
    try {
      BufferedInputStream in =
        new BufferedInputStream(new FileInputStream(file));
      ByteArrayOutputStream byteStream =
        new ByteArrayOutputStream(512);
      int imageByte;
```

Listing 7.7 ImageRetriever.java (continued)

```
      while((imageByte = in.read()) != -1) {
        byteStream.write(imageByte);
      }
      in.close();
      String persistenceFlag =
      request.getParameter("usePersistence");
      boolean usePersistence =
        ((persistenceFlag == null) ||
         (!persistenceFlag.equals("no")));
      response.setContentType("image/gif");
      if (usePersistence) {
        response.setContentLength(byteStream.size());
      }
      byteStream.writeTo(response.getOutputStream());
    } catch(IOException ioe) {
      reportError(response, "Error: " + ioe);
    }
  }

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

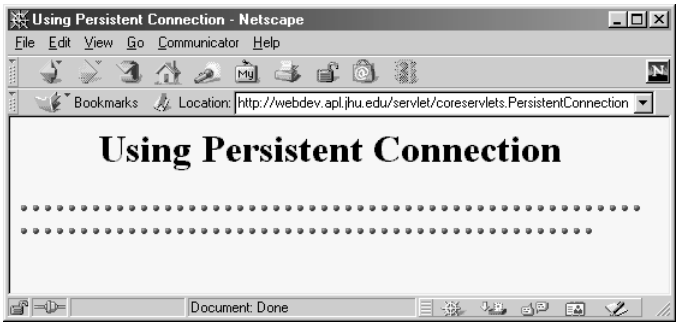


Figure 7–4 Result of the `PersistentConnection` servlet.

7.5 Using Servlets to Generate GIF Images

Although servlets often generate HTML output, they certainly don't *always* do so. For example, Section 11.2 (The contentType Attribute) shows a JSP page (which gets translated into a servlet) that builds Excel spreadsheets and returns them to the client. Here, I'll show you how to generate GIF images.

First, let me summarize the two main steps servlets have to perform in order to build multimedia content. First, they have to set the `Content-Type` response header by using the `setContentType` method of `HttpServletResponse`. Second, they have to send the output in the appropriate format. This format varies among document types, of course, but in most cases you use send binary data, not strings as 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 points 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 GIF images.

1. **Create an `Image`.**
   You create an `Image` object by using the `createImage` method of the `Component` class. Since server-side programs should not actually open any windows on the screen, they need to explicitly tell the system to create a native window system object, a process that normally occurs automatically when a window pops up. The `addNotify` method accomplishes this task. Putting this all together, here is the normal process:

   ```
   Frame f = new Frame();
   f.addNotify();
   int width = ...;
   int height = ...;
   Image img = f.createImage(width, height);
   ```

2. **Draw into the `Image`.**
   You accomplish this task by calling the `Image`'s `getGraphics` method and then using the resultant `Graphics` object in the usual manner. For example, with JDK 1.1, you would use various `drawXxx` and `fillXxx` methods of `Graphics` to draw images, strings, and shapes onto the `Image`. With the Java 2 platform, you would cast the `Graphics` object to `Graphics2D`, then make use of Java2D's much richer set of drawing operations, coordinate transformations, font settings, and fill patterns to perform the drawing. Here is a simple example:

   ```
   Graphics g = img.getGraphics();
   g.fillRect(...);
   g.drawString(...);
   ```

3. **Set the `Content-Type` response header.**
   As already discussed, you use the `setContentType` method of `HttpServletResponse` for this task. The MIME type for GIF images is `image/gif`.

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

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.

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

5. **Send the `Image` in GIF format to the output stream.**
   Accomplishing this task yourself requires quite a bit of work. Fortunately, there are several existing classes that perform this operation. One of the most popular ones is Jef Poskanzer's `GifEncoder` class, available free from `http://www.acme.com/java/`. Here is how you would use this class to send an `Image` in GIF format:

   ```
   try {
     new GifEncoder(img, out).encode();
   } catch(IOException ioe) {
     // Error message
   }
   ```

Listings 7.8 and 7.9 show a servlet that reads `message`, `fontName`, and `fontSize` parameters and uses them to create a GIF image showing the mes-

sage in the designated face and size, with a gray, oblique shadowed version of the message shown behind the main string. This operation makes use of several facilities available only in the Java 2 platform. First, it makes use of any font that is installed on the server system, rather than limiting itself to the standard names (`Serif`, `SansSerif`, `Monospaced`, `Dialog`, and `DialogInput`) available to JDK 1.1 programs.

Second, it uses the `translate`, `scale`, and `shear` transformations to create the shadowed version of the main message. Consequently, the servlet will run *only* in servlet engines running on the Java 2 platform. You would expect this to be the case with engines supporting the servlet 2.2 specification, since that is the servlet version stipulated in J2EE.

Even if you are using a server that supports only version 2.1, you should still use the Java 2 platform if you can, since it tends to be significantly more efficient for server-side tasks. However, many servlet 2.1 engines come preconfigured to use JDK 1.1, and changing the Java version is not always simple. So, for example, Tomcat and the JSWDK automatically make use of whichever version of Java is first in your `PATH`, but the Java Web Server uses a bundled version of JDK 1.1.

Listing 7.10 shows an HTML form used as a front end to the servlet. Figures 7–5 through 7–8 show some possible results. Just to simplify experimentation, Listing 7.11 presents an interactive application that lets you specify the message, font name, and font size on the command line, popping up a `JFrame` that shows the same image as the servlet would return. Figure 7–9 shows one typical result.

Listing 7.8 `ShadowedText.java`

```
package coreservlets;

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

/** Servlet that generates GIF images representing
 *  a designated message with an oblique shadowed
 *  version behind it.
 *  <P>
 *  <B>Only runs on servers that support Java 2, since
 *  it relies on Java2D to build the images.</B>
 */
public class ShadowedText extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    String message = request.getParameter("message");
    if ((message == null) || (message.length() == 0)) {
```

Listing 7.8 ShadowedText.java (continued)

```
      message = "Missing 'message' parameter";
    }
    String fontName = request.getParameter("fontName");
    if (fontName == null) {
      fontName = "Serif";
    }
    String fontSizeString = request.getParameter("fontSize");
    int fontSize;
    try {

      fontSize = Integer.parseInt(fontSizeString);
    } catch(NumberFormatException nfe) {
      fontSize = 90;
    }
    response.setContentType("image/gif");
    OutputStream out = response.getOutputStream();
    Image messageImage =
      MessageImage.makeMessageImage(message,
                                    fontName,
                                    fontSize);
    MessageImage.sendAsGIF(messageImage, out);
  }

  /** Allow form to send data via either GET or POST. */

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

Listing 7.9 MessageImage.java

```
package coreservlets;

import java.awt.*;
import java.awt.geom.*;
import java.io.*;
import Acme.JPM.Encoders.GifEncoder;

/** Utilities for building images showing shadowed messages.
 *  Includes a routine that uses Jef Poskanzer's GifEncoder
 *  to return the result as a GIF.
 *  <P>
 *  <B>Does not run in JDK 1.1, since it relies on Java2D
 *  to build the images.</B>
 *  <P>
 */
```

Listing 7.9 MessageImage.java (continued)

```
public class MessageImage {

  /** Creates an Image of a string with an oblique
   *  shadow behind it. Used by the ShadowedText servlet
   *  and the ShadowedTextFrame desktop application.
   */
  public static Image makeMessageImage(String message,
                                       String fontName,
                                       int fontSize) {
    Frame f = new Frame();
    // Connect to native screen resource for image creation.
    f.addNotify();
    // Make sure Java knows about local font names.
    GraphicsEnvironment env =
      GraphicsEnvironment.getLocalGraphicsEnvironment();
    env.getAvailableFontFamilyNames();
    Font font = new Font(fontName, Font.PLAIN, fontSize);
    FontMetrics metrics = f.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;
    Image messageImage = f.createImage(width, height);
    Graphics2D g2d =
      (Graphics2D)messageImage.getGraphics();
    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);
  }

  /** Uses GifEncoder to send the Image down output stream
   *  in GIF89A format. See http://www.acme.com/java/ for
   *  the GifEncoder class.
   */

  public static void sendAsGIF(Image image, OutputStream out) {
    try {
      new GifEncoder(image, out).encode();
    } catch(IOException ioe) {
      System.err.println("Error outputting GIF: " + ioe);
    }

  }
}
```

Listing 7.10 ShadowedText.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>GIF Generation Service</TITLE>
</HEAD>

<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN="CENTER">GIF Generation Service</H1>
Welcome to the <I>free</I> trial edition of our GIF
generation service. Enter a message, a font name,
and a font size below, then submit the form. You will
be returned a GIF 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.

<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"><BR><BR>
    <Input TYPE="SUBMIT" VALUE="Build Image">
  </CENTER>
</FORM>

</BODY>
</HTML>
```

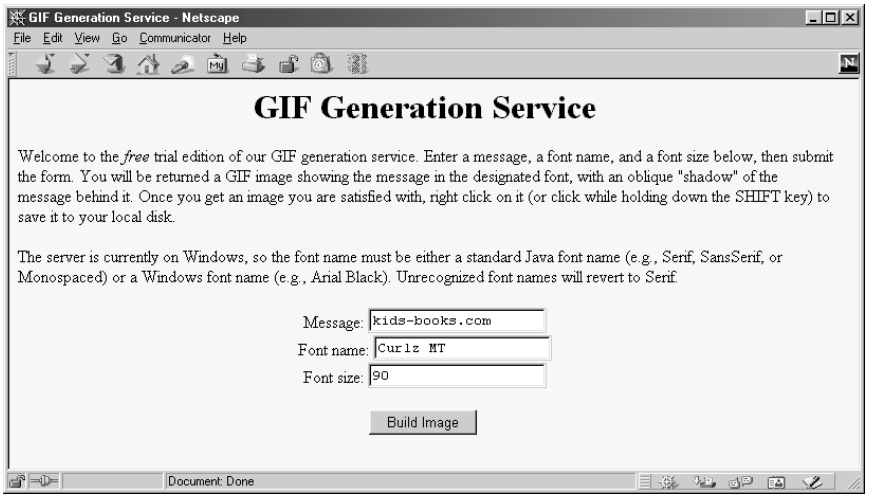


Figure 7–5 Front end to `ShadowedText` servlet.

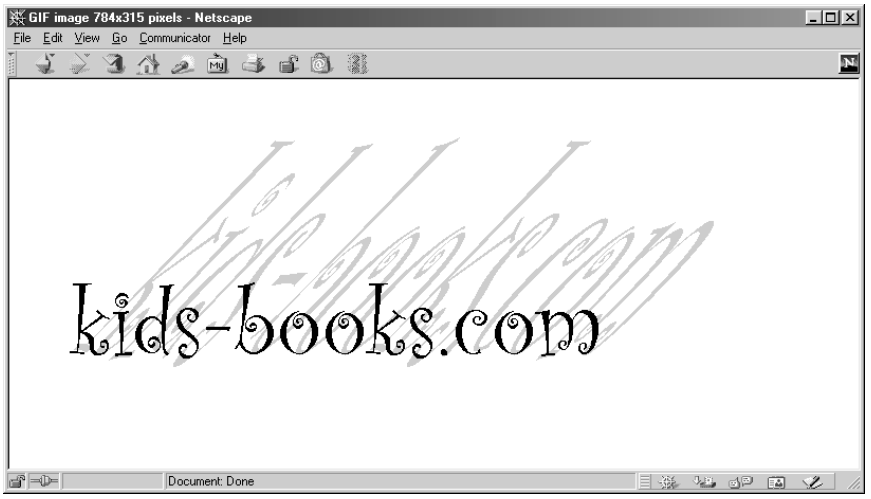


Figure 7–6 Using the GIF-generation servlet to build the logo for a children's books Web site. (Result of submitting the form shown in Figure 7–5).

Figure 7–7 Using the GIF-generation servlet to build the title image for a site describing a local theater company.

Figure 7–8 Using the GIF-generation servlet to build an image for a page advertising a local carnival.

Listing 7.11 ShadowedTextFrame.java

```
package coreservlets;

import java.awt.*;
import javax.swing.*;
import java.awt.geom.*;

/** Interactive interface to MessageImage class.
 *  Enter message, font name, and font size on the command
 *  line. Requires Java2.
 */

public class ShadowedTextFrame extends JPanel {
  private Image messageImage;

  public static void main(String[] args) {
    String message = "Shadowed Text";
    if (args.length > 0) {
      message = args[0];
    }
    String fontName = "Serif";
    if (args.length > 1) {
      fontName = args[1];
    }
    int fontSize = 90;
    if (args.length > 2) {
      try {
        fontSize = Integer.parseInt(args[2]);
      } catch(NumberFormatException nfe) {}
    }
    JFrame frame = new JFrame("Shadowed Text");
    frame.addWindowListener(new ExitListener());
    JPanel panel =
      new ShadowedTextFrame(message, fontName, fontSize);
    frame.setContentPane(panel);
    frame.pack();
    frame.setVisible(true);
  }

  public ShadowedTextFrame(String message,
                           String fontName,
                           int fontSize) {
    messageImage = MessageImage.makeMessageImage(message,
                                                 fontName,
                                                 fontSize);
    int width = messageImage.getWidth(this);
    int height = messageImage.getHeight(this);
    setPreferredSize(new Dimension(width, height));
  }
```

Listing 7.11 ShadowedTextFrame.java (continued)

```
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.drawImage(messageImage, 0, 0, this);
  }
}
```

Listing 7.12 ExitListener.java

```
package coreservlets;

import java.awt.*;
import java.awt.event.*;

/** A listener that you attach to the top-level Frame or JFrame
 *  of your application, so quitting the frame exits the app.
 */

public class ExitListener extends WindowAdapter {
  public void windowClosing(WindowEvent event) {
    System.exit(0);
  }
}
```

Figure 7–9 ShadowedTextFrame application when invoked with "java coreservlets.ShadowedTextFrame "Tom's Tools" Haettenschweiler 100".