Table of Contents

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Chapter 15. URLConnections

`URLConnection` is an abstract class that represents an active connection to a resource specified by a URL. The `URLConnection` class has two different but related purposes. First, it provides more control over the interaction with a server (especially an HTTP server) than the `URL` class. With a `URLConnection`, you can inspect the header sent by the server and respond accordingly. You can set the header fields used in the client request. You can use a `URLConnection` to download binary files. Finally, a `URLConnection` lets you send data back to a web server with POST or PUT and use other HTTP request methods. We will explore all of these techniques in this chapter.

Second, the `URLConnection` class is part of Java's *protocol handler* mechanism, which also includes the `URLStreamHandler` class. The idea behind protocol handlers is simple: they separate the details of processing a protocol from processing particular data types, providing user interfaces, and doing the other work that a monolithic web browser performs. The base `java.net.URLConnection` class is abstract; to implement a specific protocol, you write a subclass. These subclasses can be loaded at runtime by applications. For example, if the browser runs across a URL with a strange scheme, such as *compress*, rather than throwing up its hands and issuing an error message, it can download a protocol handler for this unknown protocol and use it to communicate with the server. Writing protocol handlers is the subject of the next chapter.

Only abstract `URLConnection` classes are present in the `java.net` package. The concrete subclasses are hidden inside the `sun.net` package hierarchy. Many of the methods and fields as well as the single constructor in the `URLConnection` class are *protected*. In other words, they can only be accessed by instances of the `URLConnection` class or its subclasses. It is rare to instantiate `URLConnection` objects directly in your source code; instead, the runtime environment creates these objects as needed, depending on the protocol in use. The class (which is unknown at compile time) is then instantiated using the `forName()` and `newInstance( )` methods of the `java.lang.Class` class.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

`URLConnection`does not have the best-designed API in the Java class library. Since the `URLConnection` class itself relies on the `Socket` class for network connectivity, there's little you can do with `URLConnection` that can't also be done with `Socket`. The `URLConnection` class is supposed to provide an easier-to-use, higher-level abstraction for network connections than `Socket`. In practice, however, most programmers have chosen to ignore it and simply use the `Socket` class. One of several problems is that the `URLConnection` class is too closely tied to the HTTP protocol. For instance, it assumes that each file transferred is preceded by a MIME header or something very much like one. However, most classic protocols such as FTP and SMTP don't use MIME headers. Another problem, one I hope to alleviate in this chapter, is that the `URLConnection` class is extremely poorly documented, so very few programmers understand how it's really supposed to work.

15.1. Opening URLConnections

A program that uses the `URLConnection` class directly follows this basic sequence of steps:

1. Construct a `URL` object.
2. Invoke the `URL` object's `openConnection( )` method to retrieve a `URLConnection` object for that URL.
3. Configure the `URLConnection`.
4. Read the header fields.
5. Get an input stream and read data.
6. Get an output stream and write data.
7. Close the connection.

You don't always perform all these steps. For instance, if the default setup for a particular kind of URL is acceptable, then you're likely to skip step 3. If you only want the data from the server and don't care about any metainformation, or if the protocol doesn't provide any metainformation, you'll skip step 4. If you only want to receive data from the server but not send data to the server, you'll skip step 6. Depending on the protocol, steps 5 and 6 may be reversed or interlaced.

The single constructor for the `URLConnection` class is protected:

```
protected URLConnection(URL url)
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Consequently, unless you're subclassing `URLConnection` to handle a new kind of URL (that is, writing a protocol handler), you can only get a reference to one of these objects through the `openConnection( )` methods of the `URL` and `URLStreamHandler` classes. For example:

```
try {
  URL u = new URL("http://www.greenpeace.org/");
  URLConnection uc = u.openConnection( );
}
catch (MalformedURLException ex) {
  System.err.println(ex);
}
catch (IOException ex) {
  System.err.println(ex);
}
```

In practice, the `openConnection( )` method of `java.net.URL` is the same as the `openConnection( )` method of `java.net.URLStreamHandler`. All a `URL` object's `openConnection( )` method does is call its `URLStreamHandler`'s `openConnection( )` method.

The `URLConnection` class is declared abstract. However, all but one of its methods are implemented. You may find it convenient or necessary to override other methods in the class; but the single method that subclasses must implement is `connect()`, which makes a connection to a server and thus depends on the type of service (HTTP, FTP, and so on). For example, a `sun.net.www.protocol.file.FileURLConnection`'s `connect( )` method converts the URL to a filename in the appropriate directory, creates MIME information for the file, and then opens a buffered `FileInputStream` to the file. The `connect( )` method of `sun.net.www.protocol.http.HttpURLConnection` creates a `sun.net.www.http.HttpClient` object, which is responsible for connecting to the server.

```
public abstract void connect( ) throws IOException
```

When a `URLConnection` is first constructed, it is unconnected; that is, the local and remote host cannot send and receive data. There is no socket connecting the two hosts. The `connect( )` method establishes a connection—normally using TCP sockets but possibly through some other mechanism—between the local and remote host so they can send and receive data. However, `getInputStream( )`, `getContent( )`, `getHeaderField( )`, and other methods that require an open connection will call `connect( )` if the connection isn't yet open. Therefore, you rarely need to call `connect( )` directly.

15.2. Reading Data from a Server

Here is the minimal set of steps needed to retrieve data from a URL using a `URLConnection` object:

1. Construct a `URL` object.
2. Invoke the `URL` object's `openConnection( )` method to retrieve a `URLConnection` object for that URL.
3. Invoke the `URLConnection`'s `getInputStream( )` method.
4. Read from the input stream using the usual stream API.

The `getInputStream()` method returns a generic `InputStream` that lets you read and parse the data that the server sends.

```
public InputStream getInputStream( )
```

Example 15-1 uses the `getInputStream()` method to download a web page.

Example 15-1. Download a web page with a URLConnection

```
import java.net.*;
import java.io.*;

public class SourceViewer2 {

  public static void main (String[] args) {

    if  (args.length > 0) {
      try {
        //Open the URLConnection for reading
        URL u = new URL(args[0]);
        URLConnection uc = u.openConnection( );
        InputStream raw = uc.getInputStream( );
        InputStream buffer = new BufferedInputStream(raw);
        // chain the InputStream to a Reader
        Reader r = new InputStreamReader(buffer);
        int c;
        while ((c = r.read( )) != -1) {
          System.out.print((char) c);
        }
      }
      catch (MalformedURLException ex) {
        System.err.println(args[0] + " is not a parseable URL");
      }
      catch (IOException ex) {
        System.err.println(ex);
      }
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
        } //  end if

      } // end main

    }  // end SourceViewer2
```

It is no accident that this program is almost the same as Example 7-5. The `openStream ( )` method of the `URL` class just returns an `InputStream` from its own `URLConnection` object. The output is identical as well, so I won't repeat it here.

The differences between `URL` and `URLConnection` aren't apparent with just a simple input stream as in this example. The biggest differences between the two classes are:

- `URLConnection` provides access to the HTTP header.
- `URLConnection` can configure the request parameters sent to the server.
- `URLConnection` can write data to the server as well as read data from the server.

15.3. Reading the Header

HTTP servers provide a substantial amount of information in the header that precedes each response. For example, here's a typical HTTP header returned by an Apache web server:

```
HTTP/1.1 200 OK
Date: Mon, 18 Oct 1999 20:06:48 GMT
Server: Apache/1.3.4 (Unix) PHP/3.0.6 mod_perl/1.17
Last-Modified: Mon, 18 Oct 1999 12:58:21 GMT
ETag: "1e05f2-89bb-380b196d"
Accept-Ranges: bytes
Content-Length: 35259
Connection: close
    Content-Type: text/html
```

There's a lot of information there. In general, an HTTP header may include the content type of the requested document, the length of the document in bytes, the character set in which the content is encoded, the date and time, the date the content expires, and the date the content was last modified. However, the information depends on the server; some servers send all this information for each request, others send some information, and a few don't send anything. The methods of this section allow you to query a `URLConnection` to find out what metadata the server has provided.

Aside from HTTP, very few protocols use MIME headers (and technically speaking, even the HTTP header isn't actually a MIME header; it just looks a lot like one). When writing your own

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

subclass of `URLConnection`, it is often necessary to override these methods so that they return sensible values. The most important piece of information you may be lacking is the MIME content type. `URLConnection` provides some utility methods that guess the data's content type based on its filename or the first few bytes of the data itself.

15.3.1. Retrieving Specific Header Fields

The first six methods request specific, particularly common fields from the header. These are:

- Content-type
- Content-length
- Content-encoding
- Date
- Last-modified
- Expires

15.3.1.1. public String getContentType( )

This method returns the MIME content type of the data. It relies on the web server to send a valid content type. (In a later section, we'll see how recalcitrant servers are handled.) It throws no exceptions and returns `null` if the content type isn't available. `text/html` will be the most common content type you'll encounter when connecting to web servers. Other commonly used types include `text/plain`, `image/gif`, `application/xml`, and `image/jpeg`.

If the content type is some form of text, then this header may also contain a character set part identifying the document's character encoding. For example:

```
Content-type: text/html; charset=UTF-8
```

Or:

```
Content-Type: text/xml; charset=iso-2022-jp
```

In this case, `getContentType( )` returns the full value of the Content-type field, including the character encoding. We can use this to improve on Example 15-1 by using the encoding specified in the HTTP header to decode the document, or ISO-8859-1 (the HTTP default) if no such encoding is specified. If a nontext type is encountered, an exception is thrown. Example 15-2 demonstrates:

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Example 15-2. Download a web page with the correct character set

```
import java.net.*;
import java.io.*;

public class EncodingAwareSourceViewer {

  public static void main (String[] args) {

    for (int i = 0; i < args.length; i++) {

      try {
        // set default encoding
        String encoding = "ISO-8859-1";
        URL u = new URL(args[i]);
        URLConnection uc = u.openConnection( );
        String contentType = uc.getContentType( );
        int encodingStart = contentType.indexOf("charset=");
        if (encodingStart != -1) {
            encoding = contentType.substring(encodingStart+8);
        }
        InputStream in = new BufferedInputStream(uc.getInputStream( ));
        Reader r = new InputStreamReader(in, encoding);
        int c;
        while ((c = r.read( )) != -1) {
          System.out.print((char) c);
        }
      }
      catch (MalformedURLException ex) {
        System.err.println(args[0] + " is not a parseable URL");
      }
      catch (IOException ex) {
        System.err.println(ex);
      }

    } //  end if

  } // end main

}  // end EncodingAwareSourceViewer
```

In practice, most servers don't include charset information in their Content-type headers, so this is of limited use.

15.3.1.2. public int getContentLength( )

The `getContentLength()` method tells you how many bytes there are in the content. Many servers send Content-length headers only when they're transferring a binary file, not when transferring a text file. If there is no Content-length header, `getContentLength()` returns -1. The method throws no exceptions. It is used when you need to know exactly how

many bytes to read or when you need to create a buffer large enough to hold the data in advance.

In Chapter 7, we discussed how to use the `openStream( )` method of the `URL` class to download text files from an HTTP server. Although in theory you should be able to use the same method to download a binary file, such as a GIF image or a *.class* byte code file, in practice this procedure presents a problem. HTTP servers don't always close the connection exactly where the data is finished; therefore, you don't know when to stop reading. To download a binary file, it is more reliable to use a `URLConnection`'s `getContentLength( )` method to find the file's length, then read exactly the number of bytes indicated. Example 15-3 is a program that uses this technique to save a binary file on a disk.

Example 15-3. Downloading a binary file from a web site and saving it to disk

```
import java.net.*;
import java.io.*;

public class BinarySaver {

  public static void main (String args[]) {

    for (int i = 0; i < args.length; i++) {

      try {
        URL root = new URL(args[i]);
        saveBinaryFile(root);
      }
      catch (MalformedURLException ex) {
        System.err.println(args[i] + " is not URL I understand.");
      }
      catch (IOException ex) {
        System.err.println(ex);
      }
    } // end for

  } // end main


  public static void saveBinaryFile(URL u) throws IOException {

    URLConnection uc = u.openConnection( );
    String contentType = uc.getContentType( );
    int contentLength = uc.getContentLength( );
    if (contentType.startsWith("text/") || contentLength == -1 ) {
      throw new IOException("This is not a binary file.");
    }

    InputStream raw = uc.getInputStream( );
    InputStream in  = new BufferedInputStream(raw);
    byte[] data = new byte[contentLength];
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
        int bytesRead = 0;
        int offset = 0;
        while (offset < contentLength) {
           bytesRead = in.read(data, offset, data.length-offset);
           if (bytesRead == -1) break;
           offset += bytesRead;
        }
        in.close( );

        if (offset != contentLength) {
          throw new IOException("Only read " + offset
           + " bytes; Expected " + contentLength + " bytes");
        }

        String filename = u.getFile( );
        filename = filename.substring(filename.lastIndexOf('/') + 1);
        FileOutputStream fout = new FileOutputStream(filename);
        fout.write(data);
        fout.flush( );
        fout.close( );

      }

    } // end BinarySaver
```

As usual, the `main( )` method loops over the URLs entered on the command line, passing each URL to the `saveBinaryFile( )` method. `saveBinaryFile()` opens a `URLConnection uc` to the `URL`. It puts the type into the variable `contentType` and the content length into the variable `contentLength`. Next, an `if` statement checks whether the content type is `text` or the Content-length field is missing or invalid (`contentLength == -1`). If either of these is `true`, an `IOException` is thrown. If these assertions are both `false`, we have a binary file of known length: that's what we want.

Now that we have a genuine binary file on our hands, we prepare to read it into an array of bytes called `data`. `data` is initialized to the number of bytes required to hold the binary object, `contentLength`. Ideally, you would like to fill `data` with a single call to `read( )` but you probably won't get all the bytes at once, so the read is placed in a loop. The number of bytes read up to this point is accumulated into the `offset` variable, which also keeps track of the location in the `data` array at which to start placing the data retrieved by the next call to `read( )`. The loop continues until `offset` equals or exceeds `contentLength`; that is, the array has been filled with the expected number of bytes. We also break out of the `while` loop if `read( )` returns -1, indicating an unexpected end of stream. The `offset` variable now contains the total number of bytes read, which should be equal to the content length. If they are not equal, an error has occurred, so `saveBinaryFile()` throws an `IOException`. This is the general procedure for reading binary files from HTTP connections.

Now we are ready to save the data in a file. `saveBinaryFile()` gets the filename from the URL using the `getFile( )` method and strips any path information by calling

`filename.substring(theFile.lastIndexOf('/') + 1)`. A new `FileOutputStream fout` is opened into this file and the data is written in one large burst with `fout.write(b)`.

15.3.1.3. public String getContentEncoding( )

This method returns a `String` that tells you how the content is encoded. If the content is sent unencoded (as is commonly the case with HTTP servers), this method returns `null`. It throws no exceptions. The most commonly used content encoding on the Web is probably x-gzip, which can be straightforwardly decoded using a `java.util.zip.GZipInputStream`.

The content encoding is not the same as the character encoding. The character encoding is determined by the Content-type header or information internal to the document, and specifies how characters are specified in bytes. Content encoding specifies how the bytes are encoded in other bytes.

When subclassing `URLConnection`, override this method if you expect to be dealing with encoded data, as might be the case for an NNTP or SMTP protocol handler; in these applications, many different encoding schemes, such as BinHex and uuencode, are used to pass eight-bit binary data through a seven-bit ASCII connection.

15.3.1.4. public long getDate( )

The `getDate( )` method returns a `long` that tells you when the document was sent, in milliseconds since midnight, Greenwich Mean Time (GMT), January 1, 1970. You can convert it to a `java.util.Date`. For example:

```
Date documentSent = new Date(uc.getDate( ));
```

This is the time the document was sent as seen from the server; it may not agree with the time on your local machine. If the HTTP header does not include a Date field, `getDate ( )` returns 0.

15.3.1.5. public long getExpiration( )

Some documents have server-based expiration dates that indicate when the document should be deleted from the cache and reloaded from the server. `getExpiration( )` is very similar to `getDate( )`, differing only in how the return value is interpreted. It returns a `long` indicating the number of milliseconds after 12:00 A.M., GMT, January 1, 1970, at which point the document expires. If the HTTP header does not include an Expiration field, `getExpiration( )` returns 0, which means 12:00 A.M., GMT, January 1, 1970. The only reasonable interpretation of this date is that the document does not expire and can remain in the cache indefinitely.

15.3.1.6. public long getLastModified( )

The final date method, `getLastModified( )`, returns the date on which the document was last modified. Again, the date is given as the number of milliseconds since midnight, GMT, January 1, 1970. If the HTTP header does not include a Last-modified field (and many don't), this method returns 0.

Example 15-4 reads URLs from the command line and uses these six methods to print their content type, content length, content encoding, date of last modification, expiration date, and current date.

Example 15-4. Return the header

```
import java.net.*;
import java.io.*;
import java.util.*;

public class HeaderViewer {

  public static void main(String args[]) {

    for (int i=0; i < args.length; i++) {
      try {
        URL u = new URL(args[0]);
        URLConnection uc = u.openConnection( );
        System.out.println("Content-type: " + uc.getContentType( ));
        System.out.println("Content-encoding: "
         + uc.getContentEncoding( ));
        System.out.println("Date: " + new Date(uc.getDate( )));
        System.out.println("Last modified: "
         + new Date(uc.getLastModified( )));
        System.out.println("Expiration date: "
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
            + new Date(uc.getExpiration( )));
          System.out.println("Content-length: " + uc.getContentLength( ));
        }  // end try
        catch (MalformedURLException ex) {
          System.err.println(args[i] + " is not a URL I understand");
        }
        catch (IOException ex) {
          System.err.println(ex);
        }
        System.out.println( );
      }  // end for

    }  // end main

  }  // end HeaderViewer
```

Here's the result when used to look at http://www.oreilly.com:

```
% java HeaderViewer http://www.oreilly.com
Content-type: text/html
Content-encoding: null
Date: Mon Oct 18 13:54:52 PDT 1999
Last modified: Sat Oct 16 07:54:02 PDT 1999
Expiration date: Wed Dec 31 16:00:00 PST 1969
Content-length: -1
```

The content type of the file at http://www.oreilly.com is `text/html`. No content encoding was used. The file was sent on Monday, October 18, 1999 at 1:54 P.M., Pacific Daylight Time. It was last modified on Saturday, October 16, 1999 at 7:54 A.M. Pacific Daylight Time and it expires on Wednesday, December 31, 1969 at 4:00 P. M., Pacific Standard Time. Did this document really expire 31 years ago? No. Remember that what's being checked here is whether the copy in your cache is more recent than 4:00 P.M. PST, December 31, 1969. If it is, you don't need to reload it. More to the point, after adjusting for time zone differences, this date looks suspiciously like 12:00 A.M., Greenwich Mean Time, January 1, 1970, which happens to be the default if the server doesn't send an expiration date. (Most don't.)

Finally, the content length of -1 means that there was no Content-length header. Many servers don't bother to provide a Content-length header for text files. However, a Content-length header should always be sent for a binary file. Here's the HTTP header you get when you request the GIF image http://www.oreilly.com/graphics/space.gif. Now the server sends a Content-length header with a value of 57.

```
% java HeaderViewer http://www.oreilly.com/graphics/space.gif
Content-type: image/gif
Content-encoding: null
Date: Mon Oct 18 14:00:07 PDT 1999
Last modified: Thu Jan 09 12:05:11 PST 1997
Expiration date: Wed Dec 31 16:00:00 PST 1969
Content-length: 57
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

15.3.2. Retrieving Arbitrary Header Fields

The last six methods requested specific fields from the header, but there's no theoretical limit to the number of header fields a message can contain. The next five methods inspect arbitrary fields in a header. Indeed, the methods of the last section are just thin wrappers over the methods discussed here; you can use these methods to get header fields that Java's designers did not plan for. If the requested header is found, it is returned. Otherwise, the method returns `null`.

15.3.2.1. public String getHeaderField(String name)

The `getHeaderField()` method returns the value of a named header field. The name of the header is not case-sensitive and does not include a closing colon. For example, to get the value of the Content-type and Content-encoding header fields of a `URLConnection` object `uc`, you could write:

```
String contentType = uc.getHeaderField("content-type");
String contentEncoding = uc.getHeaderField("content-encoding"));
```

To get the Date, Content-length, or Expires headers, you'd do the same:

```
String data = uc.getHeaderField("date");
String expires = uc.getHeaderField("expires");
String contentLength = uc.getHeaderField("Content-length");
```

These methods all return `String`, not `int` or `long` as the `getContentLength( )`, `getExpirationDate(),getLastModified( )`, and `getDate( )` methods of the last section did. If you're interested in a numeric value, convert the `String` to a `long` or an `int`.

Do not assume the value returned by `getHeaderField()` is valid. You must check to make sure it is non-null.

15.3.2.2. public String getHeaderFieldKey(int n)

This method returns the key (that is, the field name: for example, `Content-length` or `Server`) of the n^{th} header field. The request method is header zero and has a null key. The first header is one. For example, to get the sixth key of the header of the `URLConnection uc`, you would write:

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
String header6 = uc.getHeaderFieldKey(6);
```

15.3.2.3. public String getHeaderField(int n)

This method returns the value of the *n*th header field. In HTTP, the request method is header field zero and the first actual header is one. Example 15-5 uses this method in conjunction with `getHeaderFieldKey( )` to print the entire HTTP header.

Example 15-5. Print the entire HTTP header

```
import java.net.*;
import java.io.*;

public class AllHeaders {

  public static void main(String args[]) {

    for (int i=0; i < args.length; i++) {
      try {
        URL u = new URL(args[i]);
        URLConnection uc = u.openConnection( );
        for (int j = 1; ; j++) {
          String header = uc.getHeaderField(j);
          if (header == null) break;
          System.out.println(uc.getHeaderFieldKey(j) + ": " + header);
        }  // end for
      }  // end try
      catch (MalformedURLException ex) {
        System.err.println(args[i] + " is not a URL I understand.");
      }
      catch (IOException ex) {
        System.err.println(ex);
      }
      System.out.println( );
    }  // end for

  }  // end main

}  // end AllHeaders
```

For example, here's the output when this program is run against http://www.oreilly.com:

```
% java AllHeaders http://www.oreilly.com
Server: WN/1.15.1
Date: Mon, 18 Oct 1999 21:20:26 GMT
Last-modified: Sat, 16 Oct 1999 14:54:02 GMT
Content-type: text/html
Title: www.oreilly.com -- Welcome to O'Reilly &amp; Associates!
-- computer  books, software, online publishing
Link: <mailto:webmaster@oreilly.com>; rev="Made"
```

Besides Date, Last-modified, and Content-type headers, this server also provides Server, Title, and Link headers. Other servers may have different sets of headers.

15.3.2.4. public long getHeaderFieldDate(String name, long default)

This method first retrieves the header field specified by the `name` argument and tries to convert the string to a `long` that specifies the milliseconds since midnight, January 1, 1970, GMT. `getHeaderFieldDate()` can be used to retrieve a header field that represents a date: for example, the Expires, Date, or Last-modified headers. To convert the string to an integer, `getHeaderFieldDate()` uses the `parseDate( )` method of `java.util.Date`. The `parseDate()` method does a decent job of understanding and converting most common date formats, but it can be stumped—for instance, if you ask for a header field that contains something other than a date. If `parseDate( )` doesn't understand the date or if `getHeaderFieldDate( )` is unable to find the requested header field, `getHeaderFieldDate( )` returns the `default` argument. For example:

```
Date expires = new Date(uc.getHeaderFieldDate("expires", 0));
long lastModified = uc.getHeaderFieldDate("last-modified", 0);
Date now = new Date(uc.getHeaderFieldDate("date", 0));
```

You can use the methods of the `java.util.Date` class to convert the `long` to a `String`.

15.3.2.5. public int getHeaderFieldInt(String name, int default)

This method retrieves the value of the header field `name` and tries to convert it to an `int`. If it fails, either because it can't find the requested header field or because that field does not contain a recognizable integer, `getHeaderFieldInt( )` returns the `default` argument. This method is often used to retrieve the `Content-length` field. For example, to get the content length from a `URLConnection uc`, you would write:

```
int contentLength = uc.getHeaderFieldInt("content-length", -1);
```

In this code fragment, `getHeaderFieldInt( )` returns -1 if the Content-length header isn't present.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

15.4. Configuring the Connection

The `URLConnection` class has seven protected instance fields that define exactly how the client makes the request to the server. These are:

```
protected URL     url;
protected boolean doInput = true;
protected boolean doOutput = false;
protected boolean allowUserInteraction = defaultAllowUserInteraction;
protected boolean useCaches = defaultUseCaches;
protected long    ifModifiedSince = 0;
protected boolean connected = false;
```

For instance, if `doOutput` is `true`, you'll be able to write data to the server over this `URLConnection` as well as read data from it. If `useCaches` is `false`, the connection bypasses any local caching and downloads the file from the server afresh.

Since these fields are all protected, their values are accessed and modified via obviously named setter and getter methods:

```
public URL     getURL( )
public void    setDoInput(boolean doInput)
public boolean getDoInput( )
public void    setDoOutput(boolean doOutput)
public boolean getDoOutput( )
public void    setAllowUserInteraction(boolean allowUserInteraction)
public boolean getAllowUserInteraction( )
public void    setUseCaches(boolean useCaches)
public boolean getUseCaches( )
public void    setIfModifiedSince(long ifModifiedSince)
public long    getIfModifiedSince( )
```

You can modify these fields only before the `URLConnection` is connected (that is, before you try to read content or headers from the connection). Most of the methods that set fields throw an `IllegalStateException` if they are called while the connection is open. In general, you can set the properties of a `URLConnection` object only before the connection is opened.

In Java 1.3 and earlier, the setter methods throw an `IllegalAccessError` instead of an `IllegalStateException`. Throwing an *error* instead of an *exception* here is very unusual. An error generally indicates an unpredictable fault in the VM, which usually

cannot be handled, whereas an exception indicates a predictable, manageable problem. More specifically, an `IllegalAccessError` is supposed to indicate that an application is trying to access a nonpublic field it doesn't have access to. According to the class library documentation, "Normally, this error is caught by the compiler; this error can only occur at runtime if the definition of a class has incompatibly changed." Clearly, that's not what's going on here. This was simply a mistake on the part of the programmer who wrote this class, which has been fixed as of Java 1.4.

There are also some getter and setter methods that define the default behavior for all instances of `URLConnection`. These are:

```
public boolean            getDefaultUseCaches( )
public void               setDefaultUseCaches(boolean defaultUseCaches)
public static void        setDefaultAllowUserInteraction(
   boolean defaultAllowUserInteraction)
public static boolean     getDefaultAllowUserInteraction( )
public static FileNameMap getFileNameMap( )
public static void        setFileNameMap(FileNameMap map)
```

Unlike the instance methods, these methods can be invoked at any time. The new defaults will apply only to `URLConnection` objects constructed after the new default values are set.

15.4.1. protected URL url

The `url` field specifies the URL that this `URLConnection` connects to. The constructor sets it when the `URLConnection` is created and it should not change thereafter. You can retrieve the value by calling the `getURL( )` method. Example 15-6 opens a `URLConnection` to http://www.oreilly.com/, gets the `URL` of that connection, and prints it.

Example 15-6. Print the URL of a URLConnection to http://www.oreilly.com/

```
import java.net.*;
import java.io.*;

public class URLPrinter {
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
    public static void main(String args[]) {

      try {
        URL u = new URL("http://www.oreilly.com/");
        URLConnection uc = u.openConnection( );
        System.out.println(uc.getURL( ));
      }
      catch (IOException ex) {
        System.err.println(ex);
      }

    }

}
```

Here's the result, which should be no great surprise. The URL that is printed is the one used to create the `URLConnection`.

```
% java URLPrinter
http://www.oreilly.com/
```

15.4.2. protected boolean connected

The boolean field `connected` is `true` if the connection is open and `false` if it's closed. Since the connection has not yet been opened when a new `URLConnection` object is created, its initial value is `false`. This variable can be accessed only by instances of `java.net.URLConnection` and its subclasses.

There are no methods that directly read or change the value of `connected`. However, any method that causes the `URLConnection` to connect should set this variable to true, including `connect( )`, `getInputStream( )`, and `getOutputStream( )`. Any method that causes the `URLConnection` to disconnect should set this field to false. There are no such methods in `java.net.URLConnection`, but some of its subclasses, such as `java.net.HttpURLConnection`, have `disconnect( )` methods.

If you subclass `URLConnection` to write a protocol handler, you are responsible for setting `connected` to `true` when you are connected and resetting it to `false` when the connection closes. Many methods in `java.net.URLConnection` read this variable to determine what they can do. If it's set incorrectly, your program will have severe bugs that are not easy to diagnose.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

15.4.3. protected boolean allowUserInteraction

Some `URLConnection`s need to interact with a user. For example, a web browser may need to ask for a username and password. However, many applications cannot assume that a user is present to interact with it. For instance, a search engine robot is probably running in the background without any user to provide a username and password. As its name suggests, the `allowUserInteraction` field specifies whether user interaction is allowed. It is `false` by default.

This variable is protected, but the public `getAllowUserInteraction()` method can read its value and the public `setAllowUserInteraction()` method can change it:

```
public void    setAllowUserInteraction(boolean allowUserInteraction)
public boolean getAllowUserInteraction( )
```

The value `true` indicates that user interaction is allowed; `false` indicates that there is no user interaction. The value may be read at any time but may be set only before the `URLConnection` is connected. Calling `setAllowUserInteraction( )` when the `URLConnection` is connected throws an `IllegalStateException` in Java 1.4 and later, and an `IllegalAccessError` in Java 1.3 and earlier.

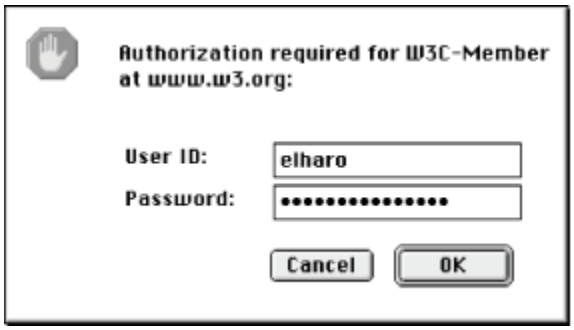
For example, this code fragment opens a connection that could ask the user for authentication if it's required:

```
URL u = new URL("http://www.example.com/passwordProtectedPage.html");
URLConnection uc = u.openConnection( );
uc.setAllowUserInteraction(true);
InputStream in = uc.getInputStream( );
```

Java does not include a default GUI for asking the user for a username and password. If the request is made from an applet, the browser's usual authentication dialog can be relied on. In a standalone application, you first need to install an `Authenticator`, as discussed in Chapter 7.

Figure 15-1 shows the dialog box that pops up when you try to access a password-protected page. If you cancel this dialog, you'll get a 401 Authorization Required error and whatever text the server sends to unauthorized users. However, if you refuse to send authorization at all—which you can do by pressing OK, then answering No when asked if you want to retry authorization—`getInputStream( )` will throw a `ProtocolException`.

Figure 15-1. An authentication dialog



The static `getDefaultAllowUserInteraction()` and `setDefaultAllowUserInteraction( )` methods determine the default behavior for `URLConnection` objects that have not set `allowUserInteraction` explicitly. Since the `allowUserInteraction` field is `static` (i.e., a class variable instead of an instance variable), setting it changes the default behavior for all instances of the `URLConnection` class that are created after `setDefaultAllowUserInteraction( )` is called.

For instance, the following code fragment checks to see whether user interaction is allowed by default with `getDefaultAllowUserInteraction( )`. If user interaction is not allowed by default, the code uses `setDefaultAllowUserInteraction( )` to make allowing user interaction the default behavior.

```
if (!URLConnection.getDefaultAllowUserInteraction( )) {
  URLConnection.setDefaultAllowUserInteraction(true);
}
```

15.4.4. protected boolean doInput

Most `URLConnection` objects provide input to a client program. For example, a connection to a web server with the GET method would produce input for the client. However, a connection to a web server with the POST method might not. A `URLConnection` can be used for input to the program, output from the program, or both. The protected boolean field `doInput` is `true` if the `URLConnection` can be used for input, `false` if it cannot be. The default is `true`. To access this protected variable, use the public `getDoInput()` and `setDoInput()` methods:

```
public void    setDoInput(boolean doInput)
public boolean getDoInput( )
```

For example:

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
try {
  URL u = new URL("http://www.oreilly.com");
  URLConnection uc = u.openConnection( );
  if (!uc.getDoInput( )) {
    uc.setDoInput(true);
  }
  // read from the connection...
catch (IOException ex) {
  System.err.println(ex);
}
```

15.4.5. protected boolean doOutput

Programs can use a `URLConnection` to send output back to the server. For example, a program that needs to send data to the server using the POST method could do so by getting an output stream from a `URLConnection`. The protected boolean field `doOutput` is `true` if the `URLConnection` can be used for output, `false` if it cannot be; it is `false` by default. To access this protected variable, use the `getDoOutput( )` and `setDoOutput ( )` methods:

```
public void    setDoOutput(boolean dooutput)
public boolean getDoOutput( )
```

For example:

```
try {
  URL u = new URL("http://www.oreilly.com");
  URLConnection uc = u.openConnection( );
  if (!uc.getDoOutput( )) {
    uc.setDoOutput(true);
  }
  // write to the connection...
catch (IOException ex) {
  System.err.println(ex);
}
```

When you set `doOutput` to true for an *http* URL, the request method is changed from GET to POST. In Chapter 7, you saw how to send data to server-side programs with GET. GET is straightforward to work with, but its use should be limited to "safe" operations: operations that don't commit the user or have obvious side effects. For instance, it would be inappropriate to use GET to complete a purchase or add an item to a shopping cart, but you could use GET to search for the items before placing them in the cart. Unsafe operations, which should not be bookmarked or cached, should use POST (or occasionally PUT or DELETE) instead. We'll explore this in more detail later in this chapter when we talk about writing data to a server.

In earlier editions of this book, I suggested using the POST method in preference to GET for long (greater than 255 characters) URLs since some browsers had limits on the maximum length of a URL they could safely handle. In 2004, this is only really an issue with very old browsers no one is likely to be using anymore. I was planning not to even mention this issue in this chapter; but as I worked on an unrelated project during the revision of this chapter, I encountered a *server-side* limitation on URL size while writing a PHP script to process a form. I had over a thousand different fields in a form (a checklist of bird species found in New York City along with observation notes) and over 10K of data in each request. The browser handled the long URL with aplomb. However, faced with such an extreme case, the server refused to process the request until I switched from GET to POST. Thus for *very* long URLs, POST may still be necessary, even for safe operations. Alternately, you could fix the server so it doesn't object to long URLs; but for those of us who don't manage our own servers, this may not always be an option.

15.4.6. protected boolean ifModifiedSince

Many clients, especially web clients, keep caches of previously retrieved documents. If the user asks for the same document again, it can be retrieved from the cache. However, it may have changed on the server since it was last retrieved. The only way to tell is to ask the server. Clients can include an If-Modified-Since in the client request HTTP header. This header includes a date and time. If the document has changed since that time, the server should send it. Otherwise, it should not. Typically, this time is the last time the client fetched the document. For example, this client request says the document should be returned only if it has changed since 7:22:07 A.M., October 31, 2004, Greenwich Mean Time:

```
GET / HTTP/1.1
User-Agent: Java/1.4.2_05
Host: login.metalab.unc.edu:56452
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: close
If-Modified-Since: Sun, 31 Oct 2004 19:22:07 GMT
```

If the document has changed since that time, the server will send it as usual. Otherwise, it replies with a 304 Not Modified message, like this:

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
HTTP/1.0 304 Not Modified
Server: WN/1.15.1
Date: Tue, 02 Nov 2004 16:26:16 GMT
Last-modified: Fri, 29 Oct 2004 23:40:06 GMT
```

The client then loads the document from its cache. Not all web servers respect the If-Modified-Since field. Some will send the document whether it's changed or not.

The `ifModifiedSince` field in the `URLConnection` class specifies the date (in milliseconds since midnight, Greenwich Mean Time, January 1, 1970), which will be placed in the If-Modified-Since header field. Because `ifModifiedSince` is `protected`, programs should call the `getIfModifiedSince( )` and `setIfModifiedSince( )` methods to read or modify it:

```
public long getIfModifiedSince( )
public void setIfModifiedSince(long ifModifiedSince)
```

Example 15-7 prints the default value of `ifModifiedSince`, sets its value to 24 hours ago, and prints the new value. It then downloads and displays the document—but only if it's been modified in the last 24 hours.

Example 15-7. Set ifModifiedSince to 24 hours prior to now

```
import java.net.*;
import java.io.*;
import java.util.*;

public class Last24 {

  public static void main (String[] args) {

    // Initialize a Date object with the current date and time
    Date today = new Date( );
    long millisecondsPerDay = 24 * 60 * 60 * 1000;

    for (int i = 0; i < args.length; i++) {
      try {
        URL u = new URL(args[i]);
        URLConnection uc = u.openConnection( );
        System.out.println("Will retrieve file if it's modified since "
         + new Date(uc.getIfModifiedSince( )));
        uc.setIfModifiedSince((new Date(today.getTime( )
          - millisecondsPerDay)).getTime( ));
        System.out.println("Will retrieve file if it's modified since "
         + new Date(uc.getIfModifiedSince( )));
        InputStream in = new BufferedInputStream(uc.getInputStream( ));
        Reader r = new InputStreamReader(in);
        int c;
        while ((c = r.read( )) != -1) {
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
              System.out.print((char) c);
            }
            System.out.println( );

          }
          catch (Exception ex) {
            System.err.println(ex);
          }
        }

      }

    }
```

Here's the result. First, we see the default value: midnight, January 1, 1970, GMT, converted to Pacific Standard Time. Next, we see the new time, which we set to 24 hours prior to the current time:

```
% java Last24 http://www.oreilly.com
Will retrieve file if it's been modified since Wed Dec 31 16:00:00 PST 1969
Will retrieve file if it's been modified since Sun Oct 31 11:17:04 PST 2004
```

Since this document hasn't changed in the last 24 hours, it is not reprinted.

15.4.7. protected boolean useCaches

Some clients, notably web browsers, can retrieve a document from a local cache, rather than retrieving it from a server. Applets may have access to the browser's cache. Starting in Java 1.5, standalone applications can use the `java.net.ResponseCache` class described later in this chapter. The `useCaches` variable determines whether a cache will be used if it's available. The default value is `true`, meaning that the cache will be used; `false` means the cache won't be used. Because `useCaches` is `protected`, programs access it using the `getUseCaches( )` and `setUseCaches()` methods:

```
public void    setUseCaches(boolean useCaches)
public boolean getUseCaches( )
```

This code fragment disables caching to ensure that the most recent version of the document is retrieved:

```
try {
  URL u = new URL("http://www.sourcebot.com/");
  URLConnection uc = u.openConnection( );
  if (uc.getUseCaches( )) {
    uc.setUseCaches(false);
  }
```

```
}
catch (IOException ex) {
  System.err.println(ex);
}
```

Two methods define the initial value of the `useCaches` field, `getDefaultUseCaches()` and `setDefaultUseCaches( )`:

```
public void    setDefaultUseCaches(boolean useCaches)
public boolean getDefaultUseCaches( )
```

Although nonstatic, these methods do set and get a static field that determines the default behavior for all instances of the `URLConnection` class created after the change. The next code fragment disables caching by default; after this code runs, `URLConnection`s that want caching must enable it explicitly using `setUseCaches(true)`.

```
if (uc.getDefaultUseCaches( )) {
  uc.setDefaultUseCaches(false);
}
```

15.4.8. Timeouts

Java 1.5 adds four methods that allow you to query and modify the timeout values for connections; that is, how long the underlying socket will wait for a response from the remote end before throwing a `SocketTimeoutException`. These are:

```
public void setConnectTimeout(int timeout)   // Java 1.5
public int  getConnectTimeout( )              // Java 1.5
public void setReadTimeout(int timeout)      // Java 1.5
public int  getReadTimeout( )                 // Java 1.5
```

The `setConnectTimeout()`/`getConnectTimeout( )` methods control how long the socket waits for the initial connection. The `setReadTimeout()`/`getReadTimeout( )` methods control how long the input stream waits for data to arrive. All four methods measure timeouts in milliseconds. All four interpret as meaning never time out. Both setter methods throw an `IllegalArgumentException` if the timeout is negative. For example, this code fragment requests a 30-second connect timeout and a 45-second read timeout:

```
URL u = new URL("http://www.example.org");
URLConnuction uc = u.openConnection( );
uc.setConnectTimeout(30000);
uc.setReadTimeout(45000);
```

15.5. Configuring the Client Request HTTP Header

In HTTP 1.0 and later, the client sends the server not only a request line, but also a header. For example, here's the HTTP header that Wamcom Mozilla 1.3 for Mac OS uses:

```
Host: stallion.elharo.com:33119
User-Agent: Mozilla/5.0 (Macintosh; U; PPC; en-US; rv:1.3.1)
Gecko/20030723 wamcom.org
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/
plain;q=0.8,video/x-mng,image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1
Accept-Language: en-us
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Connection: close
```

A web server can use this information to serve different pages to different clients, to get and set cookies, to authenticate users through passwords, and more. Placing different fields in the header that the client sends and the server responds with does all of this.

It's important to understand that this is *not the HTTP header that the server sends to the client* and that it is read by the various `getHeaderField ( )` and `getHeaderFieldKey()` methods discussed previously. This is the *HTTP header that the client sends to the server*.

Each concrete subclass of `URLConnection` sets a number of different name-value pairs in the header by default. (Really, only `HttpURLConnection` does this, since HTTP is the only major protocol that uses headers in this way.) For instance, here's the HTTP header that a connection from the `SourceViewer2` program of Example 15-1 sends:

```
User-Agent: Java/1.4.2_05
Host: localhost:33122
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive
```

As you can see, it's a little simpler than the one Mozilla sends, and it has a different user agent and accepts different kinds of files. However, you can modify these and add new fields before connecting.

In Java 1.3 and later, you can add headers to the HTTP header using the `setRequestProperty()` method before you open the connection:

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
public void setRequestProperty(String name, String value)// Java 1.3
```

The `setRequestProperty( )` method adds a field to the header of this `URLConnection` with a specified name and value. This method can be used only before the connection is opened. It throws an `IllegalStateException` (`IllegalAccessError` in Java 1.3) if the connection is already open. The `getRequestProperty()` method returns the value of the named field of the HTTP header used by this `URLConnection`.

HTTP allows one property to have multiple values. In this case, the separate values will be separated by commas. For example, the Accept header sent by Java 1.4.2 shown above has the four values text/html, image/gif, image/jpeg, and *.

These methods only really have meaning when the URL being connected to is an *http* URL, since only the HTTP protocol makes use of headers like this. While they could possibly have other meanings in other protocols, such as NNTP, this is really just an example of poor API design. These methods should be part of the more specific `HttpURLConnection` class, not the generic `URLConnection` class.

For example, web servers and clients store some limited persistent information by using cookies. A cookie is simply a name-value pair. The server sends a cookie to a client using the response HTTP header. From that point forward, whenever the client requests a URL from that server, it includes a Cookie field in the HTTP request header that looks like this:

```
Cookie: username=elharo; password=ACD0X9F23JJJn6G; session=100678945
```

This particular Cookie field sends three name-value pairs to the server. There's no limit to the number of name-value pairs that can be included in any one cookie. Given a `URLConnection` object `uc`, you could add this cookie to the connection, like this:

```
uc.setRequestProperty("Cookie",
 "username=elharo; password=ACD0X9F23JJJn6G; session=100678945");
```

The `setRequestProperty()` method does not support this. You can set the same property to a new value, but this changes the existing property value. To add an additional property value, use the `addRequestProperty()` method instead:

```
public void  addRequestProperty(String name, String value)// Java 1.4
```

There's no fixed list of legal headers. Servers will typically ignore any headers they don't recognize. HTTP does put some restrictions on the content of the names and values here. For instance, the names can't contain whitespace and the values can't contain any line breaks. Java enforces the restrictions on fields containing line breaks, but not much else. If a field contains a line break, `setRequestProperty()` and `addRequestProperty( )` throw an `IllegalArgumentException`. Otherwise, it's quite easy to make a `URLConnection` send malformed headers to the server, so be careful. Some servers will handle the malformed headers gracefully. Some will ignore the bad header and return the requested document anyway, but some will reply with an HTTP 400, Bad Request error.

If for some reason you need to inspect the headers in a `URLConnection`, there's a standard getter method:

```
public String getRequestProperty(String name)// Java 1.3
```

Java 1.4 also adds a method to get all the request properties for a connection as a `Map`:

```
public Map getRequestProperties( ) // Java 1.4
```

The keys are the header field names. The values are lists of property values. Both names and values are stored as strings. In other words, using Java 1.5 generic syntax, the signature is:

```
public Map<String,List<String>> getRequestProperties( )
```

15.6. Writing Data to a Server

Sometimes you need to write data to a `URLConnection`—for example, when you submit a form to a web server using POST or upload a file using PUT. The `getOutputStream()` method returns an `OutputStream` on which you can write data for transmission to a server:

```
public OutputStream getOutputStream( )
```

Since a `URLConnection` doesn't allow output by default, you have to call `setDoOutput(true)` before asking for an output stream. When you set `doOutput` to true for an *http* URL, the request method is changed from GET to POST. In Chapter 7, you saw how to send data to server-side programs with GET. However, GET should be limited to safe operations, such as search requests or page navigation, and not used for unsafe operations that create or modify a resource, such as posting a comment on a web page or ordering a pizza. Safe

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

operations can be bookmarked, cached, spidered, prefetched, and so on. Unsafe operations should not be.

Once you've got the `OutputStream`, buffer it by chaining it to a `BufferedOutputStream` or a `BufferedWriter`. You should generally also chain it to a `DataOutputStream`, an `OutputStreamWriter`, or some other class that's more convenient to use than a raw `OutputStream`. For example:

```
try {

  URL u = new URL("http://www.somehost.com/cgi-bin/acgi");
  // open the connection and prepare it to POST
  URLConnection uc = u.openConnection( );
  uc.setDoOutput(true);

  OutputStream raw = uc.getOutputStream( );
  OutputStream buffered = new BufferedOutputStream(raw);
  OutputStreamWriter out = new OutputStreamWriter(buffered, "8859_1");
  out.write("first=Julie&middle=&last=Harting&work=String+Quartet\r\n");
  out.flush( );
  out.close( );

}
catch (IOException ex) {
  System.err.println(ex);
}
```

Sending data with POST is almost as easy as with GET. Invoke `setDoOutput(true)` and use the `URLConnection`'s `getOutputStream( )` method to write the query string rather than attaching it to the URL. Java buffers all the data written onto the output stream until the stream is closed. This is necessary so that it can determine the necessary Content-length header. The query string contains two name-value pairs separated by ampersands. The complete transaction, including client request and server response, looks something like this:

```
% telnet www.ibiblio.org 80
Trying 152.2.210.81...
Connected to www.ibiblio.org.
Escape character is '^]'.
POST /javafaq/books/jnp3/postquery.phtml HTTP/1.0
            ACCEPT: text/plain
            Content-type: application/x-www-form-urlencoded
            Content-length: 65
            username=Elliotte+Rusty+Harold&email=elharo%40metalab%2eunc%2eedu
HTTP/1.1 200 OK
Date: Mon, 10 May 2004 21:08:52 GMT
Server: Apache/1.3.29 (Unix) DAV/1.0.3 mod_perl/1.29 PHP/4.3.5
X-Powered-By: PHP/4.3.5
Connection: close
Content-Type: text/html

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Query Results</title>
```

```
</head>
<body>

<h1>Query Results</h1>

<p>You submitted the following name/value pairs:</p>

<ul>
<li>username = Elliotte Rusty Harold</li>
<li>email = elharo@metalab.unc.edu</li>
</ul>

<hr />
Last Modified May 10, 2004

</body>
</html>
Connection closed by foreign host.
```

For that matter, as long as you control both the client and the server, you can use any other sort of data encoding you like. For instance, SOAP and XML-RPC both POST data to web servers as XML rather than an x-www-form-url-encoded query string. However, if you deviate from the standard, you'll find that your nonconforming client can't talk to most server-side programs or that your nonconforming server-side program can't process requests from most clients. The query string format used here is used by all web browsers and is expected by most server-side APIs and tools.

Example 15-8 is a program called `FormPoster` that uses the `URLConnection` class and the `QueryString` class from Chapter 7 to post form data. The constructor sets the URL. The query string is built using the `add( )` method. The `post( )` method actually sends the data to the server by opening a `URLConnection` to the specified URL, setting its `doOutput` field to `true`, and writing the query string on the output stream. It then returns the input stream containing the server's response.

The `main( )` method is a simple test for this program that sends the name "Elliotte Rusty Harold" and the email address elharo@metalab.unc.edu to the resource at http://www.cafeaulait.org/books/jnp3/postquery.phtml. This resource is a simple form tester that accepts any input using either the POST or GET method and returns an HTML page showing the names and values that were submitted. The data returned is HTML; this example simply displays the HTML rather than attempting to parse it. It would be easy to extend this program by adding a user interface that lets you enter the name and email address to be posted—but since doing that triples the size of the program while showing nothing more of network programming, it is left as an exercise for the reader. Once you understand this example, it should be easy to write Java programs that communicate with other server-side scripts.

Example 15-8. Posting a form

```
import java.net.*;
import java.io.*;
import com.macfaq.net.*;

public class FormPoster {

  private URL url;
  // from Chapter 7, Example 7-9
  private QueryString query = new QueryString( );

  public FormPoster (URL url) {
    if (!url.getProtocol( ).toLowerCase( ).startsWith("http")) {
      throw new IllegalArgumentException(
       "Posting only works for http URLs");
    }
    this.url = url;
  }

  public void add(String name, String value) {
    query.add(name, value);
  }

  public URL getURL( ) {
    return this.url;
  }

  public InputStream post( ) throws IOException {

    // open the connection and prepare it to POST
    URLConnection uc = url.openConnection( );
    uc.setDoOutput(true);
    OutputStreamWriter out
     = new OutputStreamWriter(uc.getOutputStream( ), "ASCII");

    // The POST line, the Content-type header,
    // and the Content-length headers are sent by the URLConnection.
    // We just need to send the data
    out.write(query.toString( ));
    out.write("\r\n");
    out.flush( );
    out.close( );


    // Return the response
    return uc.getInputStream( );

  }

  public static void main(String args[]) {

    URL url;

    if (args.length > 0) {
      try {
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
            url = new URL(args[0]);
          }
          catch (MalformedURLException ex) {
            System.err.println("Usage: java FormPoster url");
            return;
          }
        }
        else {
          try {
            url = new URL(
              "http://www.cafeaulait.org/books/jnp3/postquery.phtml");
          }
          catch (MalformedURLException ex) { // shouldn't happen
            System.err.println(ex);
            return;
          }
        }

        FormPoster poster = new FormPoster(url);
        poster.add("name", "Elliotte Rusty Harold");
        poster.add("email", "elharo@metalab.unc.edu");

        try {
          InputStream in = poster.post( );

          // Read the response
          InputStreamReader r = new InputStreamReader(in);
          int c;
          while((c = r.read( )) != -1) {
            System.out.print((char) c);
          }
          System.out.println( );
          in.close( );
        }
        catch (IOException ex) {
          System.err.println(ex);
        }

      }

    }
```

Here's the response from the server:

```
% java -classpath .:jnp3e.jar FormPoster
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
        <title>Query Results</title>
</head>
<body>

<h1>Query Results</h1>

<p>You submitted the following name/value pairs:</p>

<ul>
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
<li>name = Elliotte Rusty Harold</li>
<li>email = elharo@metalab.unc.edu
</li>
</ul>

<hr />
Last Modified May 10, 2004

</body>
</html>
```

The `main( )` method tries to read the first command-line argument from `args[0]`. The argument is optional; if there is an argument, it is assumed to be a URL that can be POSTed to. If there are no arguments, `main()` initializes `url` with a default URL, http://www.cafeaulait.org/books/jnp3/postquery.phtml. `main( )` then constructs a `FormPoster` object. Two name-value pairs are added to this `FormPoster` object. Next, the `post( )` method is invoked and its response read and printed on `System.out`.

The `post( )` method is the heart of the class. It first opens a connection to the URL stored in the `url` field. It sets the `doOutput` field of this connection to `true` since this `URLConnection` needs to send output and chains the `OutputStream` for this URL to an ASCII `OutputStreamWriter` that sends the data; then flushes and closes the stream. *Do not forget to close the stream!* If the stream isn't closed, no data will be sent. Finally, the `URLConnection`'s `InputStream` is returned.

To summarize, posting data to a form requires these steps:

1. Decide what name-value pairs you'll send to the server-side program.
2. Write the server-side program that will accept and process the request. If it doesn't use any custom data encoding, you can test this program using a regular HTML form and a web browser.
3. Create a query string in your Java program. The string should look like this:

   ```
   name1=value1&name2=value2&name3=value3
   ```

   Pass each name and value in the query string to `URLEncoder.encode( )` before adding it to the query string.
4. Open a `URLConnection` to the URL of the program that will accept the data.
5. Set `doOutput` to true by invoking `setDoOutput(true)`.
6. Write the query string onto the `URLConnection`'s `OutputStream`.
7. Close the `URLConnection`'s `OutputStream`.
8. Read the server response from the `URLConnection`'s `InputStream`.

Posting forms is considerably more complex than using the GET method described in Chapter 7. However, GET should only be used for safe operations that can be bookmarked and linked to. POST should be used for unsafe operations that should not be bookmarked or linked to.

The `getOutputStream( )` method is also used for the PUT request method, a means of storing files on a web server. The data to be stored is written onto the `OutputStream` that `getOutputStream( )` returns. However, this can be done only from within the `HttpURLConnection` subclass of `URLConnection`, so discussion of PUT will have to wait a little while.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

15.7. Content Handlers

The `URLConnection` class is intimately tied to Java's protocol and content handler mechanism. The protocol handler is responsible for making connections, exchanging headers, requesting particular documents, and so forth. It handles all the overhead of the protocol for requesting files. The content handler deals only with the actual data. It takes the raw input after all headers and so forth are stripped and converts it to the right kind of object for Java to deal with; for instance, an `InputStream` or an `ImageProducer`.

15.7.1. Getting Content

The `getContent( )` methods of `URLConnection` use a content handler to turn the raw data of a connection into a Java object.

15.7.1.1. public Object getContent( ) throws IOException

This method is virtually identical to the `getContent()` method of the `URL` class. In fact, that method just calls this method. `getContent( )` downloads the object selected by the `URL` of this `URLConnection`. For `getContent()` to work, the virtual machine needs to recognize and understand the content type. The exact content types supported vary from one VM and version to the next. Sun's JDK 1.5 supports `text/plain`, `image/gif`, `image/jpeg`, `image/png`, `audio/aiff`, `audio/basic`, `audio/wav`, and a few others. Different VMs and applications may support additional types. For instance, HotJava 3.0 includes a PDF content handler. Furthermore, you can install additional content handlers that understand other content types.

`getContent( )` works only for protocols like HTTP, which has a clear understanding of MIME content types. If the content type is unknown or the protocol doesn't understand content types, `getContent( )` throws an `UnknownServiceException`.

15.7.1.2. public Object getContent(Class[] classes) throws IOException // Java 1.3

This overloaded variant of the `getContent( )` method lets you choose what class you'd like the content returned as in order to provide different object representations of data. The

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

method attempts to return the content in the form of one of the classes in the `classes` array. The order of preference is the order of the array. For instance, if you'd prefer an HTML file to be returned as a `String` but your second choice is a `Reader` and your third choice is an `InputStream`, you would write:

```
URL u = new URL("http://www.thehungersite.com/");
URLConnection uc = u.openConnection( )
Class[] types = {String.class, Reader.class, InputStream.class};
Object o = uc.getContent(types);
```

Then test for the type of the returned object using `instanceof`. For example:

```
if (o instanceof String) {
  System.out.println(o);
}
else if (o instanceof Reader) {
  int c;
  Reader r = (Reader) o;
  while ((c = r.read( )) != -1) System.out.print((char) c);
}
else if (o instanceof InputStream) {
  int c;
  InputStream in = (InputStream) o;
  while ((c = in.read( )) != -1) System.out.write(c);
}
else if (o == null) {
  System.out.println("None of the requested types were available.");
}
else {
  System.out.println("Error: unexpected type " + o.getClass( ));
}
```

That last `else` clause shouldn't be reached. If none of the requested types are available, this method is supposed to return `null` rather than returning an unexpected type.

15.7.2. ContentHandlerFactory

The `URLConnection` class contains a static `Hashtable` of `ContentHandler` objects. Whenever the `getContent( )` method of `URLConnection` is invoked, Java looks in this `Hashtable` to find the right content handler for the current URL, as indicated by the URL's Content-type. If it doesn't find a `ContentHandler` object for the MIME type, it tries to create one using a `ContentHandlerFactory` (which you'll learn more about in Chapter 17). That is, a content handler factory tells the program where it can find a content handler for a `text/html` file, an `image/gif` file, or some other kind of file. You can set the `ContentHandlerFactory` by passing an instance of the `java.net.ContentHandlerFactory` interface to the `setContentHandlerFactory( )` method:

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
public static void setContentHandlerFactory(ContentHandlerFactory factory)
 throws SecurityException, Error
```

You may set the `ContentHandlerFactory` only once per application; this method throws a generic `Error` if it is called a second time. As with most other `setFactory( )` methods, untrusted applets will generally not be allowed to set the content handler factory whether one has already been set or not. Attempting to do so throws a `SecurityException`.

15.8. The Object Methods

The `URLConnection` class overrides only one method from `java.lang.Object`, `toString( )`:

```
public String toString( )
```

Even so, there is little reason to print a `URLConnection` object or to convert one to a `String`, except perhaps if you are debugging. `toString( )` is called the same way as every other `toString( )` method.

15.9. Security Considerations for URLConnections

`URLConnection` objects are subject to all the usual security restrictions about making network connections, reading or writing files, and so forth. For instance, a `URLConnection` can be created by an untrusted applet only if the `URLConnection` is pointing to the host that the applet came from. However, the details can be a little tricky because different URL schemes and their corresponding connections can have different security implications. For example, a *jar* URL that points into the applet's own *jar* file should be fine. However, a file URL that points to a local hard drive should not be.

Before attempting to connect a URL, you may want to know whether the connection will be allowed. For this purpose, the `URLConnection` class has a `getPermission()` method:

```
public Permission getPermission( ) throws IOException// Java 1.2
```

This returns a `java.security.Permission` object that specifies what permission is needed to connect to the URL. It returns `null` if no permission is needed (e.g., there's no security manager in place). Subclasses of `URLConnection` return different subclasses of

`java.security.Permission`. For instance, if the underlying URL points to *www.gwbush.com*, `getPermission( )` returns a `java.net.SocketPermission` for the host *www.gwbush.com* with the connect and resolve actions.

15.10. Guessing MIME Content Types

If this were the best of all possible worlds, every protocol and every server would use MIME types to specify the kind of file being transferred. Unfortunately, that's not the case. Not only do we have to deal with older protocols such as FTP that predate MIME, but many HTTP servers that should use MIME don't provide MIME headers at all or lie and provide headers that are incorrect (usually because the server has been misconfigured). The `URLConnection` class provides two static methods to help programs figure out the MIME type of some data; you can use these if the content type just isn't available or if you have reason to believe that the content type you're given isn't correct. The first of these is `URLConnection.guessContentTypeFromName()`:

```
public static String guessContentTypeFromName(String name)[1]
```

This method tries to guess the content type of an object based upon the extension in the filename portion of the object's URL. It returns its best guess about the content type as a `String`. This guess is likely to be correct; people follow some fairly regular conventions when thinking up filenames.

The guesses are determined by the *content-types.properties* file, normally located in the *jre/lib* directory. On Unix, Java may also look at the *mailcap* file to help it guess. Table 15-1 shows the guesses the JDK 1.5 makes. These vary a little from one version of the JDK to the next.

Table 15-1. Java extension content-type mappings

Extension	MIME content type
No extension, or unrecognized extension	`content/unknown`
.saveme, .dump, .hqx, .arc, .o, .a, .z, .bin, .exe, .zip, .gz	`application/octet-stream`
.oda	`application/oda`

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Extension	MIME content type
.pdf	`application/pdf`
.eps, .ai, .ps	`application/postscript`
.dvi	`application/x-dvi`
.hdf	`application/x-hdf`
.latex	`application/x-latex`
.nc, .cdf	`application/x-netcdf`
.tex	`application/x-tex:`
.texinfo, .texi	`application/x-texinfo`
.t, .tr, .roff	`application/x-troff`
.man	`application/x-troff-man`
.me	`application/x-troff-me`
.ms	`application/x-troff-ms`
.src, .wsrc	`application/x-wais-source`
.zip	`application/zip`
.bcpio	`application/x-bcpio`
.cpio	`application/x-cpio`
.gtar	`application/x-gtar`
.sh, .shar	`application/x-shar`

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Extension	MIME content type
.sv4cpio	`application/x-sv4cpio:`
.sv4crc	`application/x-sv4crc`
.tar	`application/x-tar`
.ustar	`application/x-ustar`
.snd, .au	`audio/basic`
.aifc, .aif, .aiff	`audio/x-aiff`
.wav	`audio/x-wav`
.gif	`image/gif`
.ief	`image/ief`
.jfif, .jfif-tbnl, .jpe, .jpg, .jpeg	`image/jpeg`
.tif, .tiff	`image/tiff`
.fpx, .fpix	`image/vnd.fpx`
.ras	`image/x-cmu-rast`
.pnm	`image/x-portable-anymap`
.pbm	`image/x-portable-bitmap`
.pgm	`image/x-portable-graymap`
.ppm	`image/x-portable-pixmap`

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Extension	MIME content type
.rgb	`image/x-rgb`
.xbm, .xpm	`image/x-xbitmap`
.xwd	`image/x-xwindowdump`
.png	`image/png`
.htm, .html	`text/html`
.text, .c, .cc, .c++, .h, .pl, .txt, .java, .el	`text/plain`
.tsv	`text/tab-separated-values`
.etx	`text/x-setext`
.mpg, .mpe, .mpeg	`video/mpeg`
.mov, .qt	`video/quicktime`
.avi	`application/x-troff-msvideo`
.movie, .mv	`video/x-sgi-movie`
.mime	`message/rfc822`
.xml	`application/xml`

This list is not complete by any means. For instance, it omits various XML applications such as RDF (*.rdf*), XSL (*.xsl*), and so on that should have the MIME type `application/xml`. It also doesn't provide a MIME type for CSS stylesheets (*.css*). However, it's a good start.

The second MIME type guesser method is `URLConnection.guessContentTypeFromStream()`:

```
public static String guessContentTypeFromStream(InputStream in)
```

This method tries to guess the content type by looking at the first few bytes of data in the stream. For this method to work, the `InputStream` must support marking so that you can return to the beginning of the stream after the first bytes have been read. Java 1.5 inspects the first 11 bytes of the `InputStream`, although sometimes fewer bytes are needed to make an identification. Table 15-2 shows how Java 1.5 guesses. Note that these guesses are often not as reliable as the guesses made by the previous method. For example, an XML document that begins with a comment rather than an XML declaration would be mislabeled as an HTML file. This method should be used only as a last resort.

Table 15-2. Java first bytes content-type mappings

First bytes in hexadecimal	First bytes in ASCII	MIME content type
0xACED		`application/x-java-serialized-object`
0xCAFEBABE		`application/java-vm`
0x47494638	GIF8	`image/gif`
0x23646566	#def	`image/x-bitmap`
0x2158504D32	!XPM2	`image/x-pixmap`
0x89504E 470D0A1A0A		`image/png`
0x2E736E64		`audio/basic`
0x646E732E		`audio/basic`
0x3C3F786D6C	<?xml	`application/xml`
0xFEFF003C003F00F7		`application/xml`
0xFFFE3C003F00F700		`application/xml`
0x3C21	<!	`text/html`

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

First bytes in hexadecimal	First bytes in ASCII	MIME content type
0x3C68746D6C	<html	`text/html`
0x3C626F6479	<body	`text/html`
0x3C68656164	<head	`text/html`
0x3C48544D4C	<HTML	`text/html`
0x3C424F4459	<BODY	`text/html`
0x3C48454144	<HEAD	`text/html`
0xFFD8FFE0		`image/jpeg`
0xFFD8FFEE		`image/jpeg`
0xFFD8FFE1XXXX4578696600[2]		`image/jpeg`
0x89504E470D0A1A0A		`image/png`
0x52494646	RIFF	`audio/x-wav`
0xD0CF11E0A1B11AE1[3]		`image/vnd.fpx`

[2] The XX bytes are not checked. They can be anything.

[3] This actually just checks for a Microsoft structured storage document. Several other more complicated checks have to be made before deciding whether this is indeed an `image/vnd.fpx` document.

ASCII mappings, where they exist, are case-sensitive. For example, `guessContentTypeFromStream( )` does not recognize `<Html>` as the beginning of a `text/html` file.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

15.11. HttpURLConnection

The `java.net.HttpURLConnection` class is an abstract subclass of `URLConnection`; it provides some additional methods that are helpful when working specifically with *http* URLs:

```
public abstract class HttpURLConnection extends URLConnection
```

In particular, it contains methods to get and set the request method, decide whether to follow redirects, get the response code and message, and figure out whether a proxy server is being used. It also includes several dozen mnemonic constants matching the various HTTP response codes. Finally, it overrides the `getPermission()` method from the `URLConnection` superclass, although it doesn't change the semantics of this method at all.

Since this class is abstract and its only constructor is protected, you can't directly create instances of `HttpURLConnection`. However, if you construct a `URL` object using an *http* URL and invoke its `openConnection( )` method, the `URLConnection` object returned will be an instance of `HttpURLConnection`. Cast that `URLConnection` to `HttpURLConnection` like this:

```
URL u = new URL("http://www.amnesty.org/");
URLConnection uc = u.openConnection( );
HttpURLConnection http = (HttpURLConnection) uc;
```

Or, skipping a step, like this:

```
URL u = new URL("http://www.amnesty.org/");
HttpURLConnection http = (HttpURLConnection) u.openConnection( );
```

There's another `HttpURLConnection` class in the undocumented `sun.net.www.protocol.http` package, a concrete subclass of `java.net.HttpURLConnection` that actually implements the abstract `connect( )` method:

```
public class HttpURLConnection extends java.net.HttpURLConnection
```

There's little reason to access this class directly. It doesn't add any important methods that aren't already declared in `java.net.HttpURLConnection` or `java.net.URLConnection`. However, any `URLConnection` you open to an *http* URL will be an instance of this class.

15.11.1. The Request Method

When a web client contacts a web server, the first thing it sends is a request line. Typically, this line begins with GET and is followed by the name of the file that the client wants to retrieve and the version of the HTTP protocol that the client understands. For example:

```
GET /catalog/jfcnut/index.html HTTP/1.0
```

However, web clients can do more than simply GET files from web servers. They can POST responses to forms. They can PUT a file on a web server or DELETE a file from a server. And they can ask for just the HEAD of a document. They can ask the web server for a list of the OPTIONS supported at a given URL. They can even TRACE the request itself. All of these are accomplished by changing the request method from GET to a different keyword. For example, here's how a browser asks for just the header of a document using HEAD:

```
HEAD /catalog/jfcnut/index.html HTTP/1.1
User-Agent: Java/1.4.2_05
Host: www.oreilly.com
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: close
```

By default, `HttpURLConnection` uses the GET method. However, you can change this with the `setRequestMethod( )` method:

```
public void setRequestMethod(String method) throws ProtocolException
```

The method argument should be one of these seven case-sensitive strings:

- GET
- POST
- HEAD
- PUT
- OPTIONS

- DELETE
- TRACE

If it's some other method, then a `java.net.ProtocolException`, a subclass of `IOException`, is thrown. However, it's generally not enough to simply set the request method. Depending on what you're trying to do, you may need to adjust the HTTP header and provide a message body as well. For instance, POSTing a form requires you to provide a Content-length header. We've already explored the GET and POST methods. Let's look at the other five possibilities.

Some web servers support additional, nonstandard request methods. For instance, Apache 1.3 also supports CONNECT, PROPFIND, PROPPATCH, MKCOL, COPY, MOVE, LOCK, and UNLOCK. However, Java doesn't support any of these.

15.11.1.1. HEAD

The HEAD function is possibly the simplest of all the request methods. It behaves much like GET. However, it tells the server only to return the HTTP header, not to actually send the file. The most common use of this method is to check whether a file has been modified since the last time it was cached. Example 15-9 is a simple program that uses the HEAD request method and prints the last time a file on a server was modified.

Example 15-9. Get the time when a URL was last changed

```
import java.net.*;
import java.io.*;
import java.util.*;

 public class LastModified {

  public static void main(String args[]) {

    for (int i=0; i < args.length; i++) {
      try {
        URL u = new URL(args[i]);
        HttpURLConnection http = (HttpURLConnection) u.openConnection( );
        http.setRequestMethod("HEAD");
        System.out.println(u + "was last modified at "
         + new Date(http.getLastModified( )));
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
          }  // end try
          catch (MalformedURLException ex) {
            System.err.println(args[i] + " is not a URL I understand");
          }
          catch (IOException ex) {
            System.err.println(ex);
          }
          System.out.println( );
        }  // end for

      }  // end main

    }  // end LastModified
```

Here's the output from one run:

```
D:\JAVA\JNP3\examples\15>java LastModified http://www.ibiblio.org/xml/
http://www.ibiblio.org/xml/was last modified at Thu Aug 19 06:06:57 PDT 2004
```

It wasn't absolutely necessary to use the HEAD method here. We'd have gotten the same results with GET. But if we used GET, the entire file at http://www.ibiblio.org/xml/ would have been sent across the network, whereas all we cared about was one line in the header. When you can use HEAD, it's much more efficient to do so.

15.11.1.2. OPTIONS

The OPTIONS request method asks what options are supported for a particular URL. If the request URL is an asterisk (*), the request applies to the server as a whole rather than to one particular URL on the server. For example:

```
OPTIONS /xml/ HTTP/1.1
User-Agent: Java/1.4.2_05
Host: www.ibiblio.org
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: close
```

The server responds to an OPTIONS request by sending an HTTP header with a list of the commands allowed on that URL. For example, when the previous command was sent, here's what Apache responded:

```
Date: Thu, 21 Oct 2004 18:06:10 GMT
Server: Apache/1.3.4 (Unix) PHP/3.0.6 mod_perl/1.17
Content-Length: 0
Allow: GET, HEAD, POST, PUT, DELETE, CONNECT, OPTIONS, PATCH, PROPFIND,
PROPPATCH, MKCOL, COPY, MOVE, LOCK, UNLOCK, TRACE
Connection: close
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

The list of legal commands is found in the Allow field. However, in practice these are just the commands the server understands, not necessarily the ones it will actually perform on that URL. For instance, let's look at what happens when you try the DELETE request method.

15.11.1.3. DELETE

The DELETE method removes a file at a specified URL from a web server. Since this request is an obvious security risk, not all servers will be configured to support it, and those that are will generally demand some sort of authentication. A typical DELETE request looks like this:

```
DELETE /javafaq/2004march.html HTTP/1.1
User-Agent: Java/1.4.2_05
Host: www.ibiblio.org
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: close
```

The server is free to refuse this request or ask for identification. For example:

```
Date: Thu, 19 Aug 2004 14:32:15 GMT
Server: Apache/1.3.4 (Unix) PHP/3.0.6 mod_perl/1.17
Allow: GET, HEAD, POST, PUT, DELETE, CONNECT, OPTIONS, PATCH, PROPFIND,
PROPPATCH, MKCOL, COPY, MOVE, LOCK, UNLOCK, TRACE
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html
content-length: 313

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML><HEAD>
<TITLE>405 Method Not Allowed</TITLE>
</HEAD><BODY>
<H1>Method Not Allowed</H1>
The requested method DELETE is not allowed for the
URL /javafaq/2004march.html.<P>
<HR>
<ADDRESS>Apache/1.3.4 Server at www.ibiblio.org Port 80</ADDRESS>
</BODY></HTML>
```

Even if the server accepts this request, its response is implementation-dependent. Some servers may delete the file; others simply move it to a trash directory. Others simply mark it as not readable. Details are left up to the server vendor.

15.11.1.4. PUT

Many HTML editors and other programs that want to store files on a web server use the PUT method. It allows clients to place documents in the abstract hierarchy of the site without necessarily knowing how the site maps to the actual local filesystem. This contrasts with FTP,

where the user has to know the actual directory structure as opposed to the server's virtual directory structure.

Here's a how a browser might PUT a file on a web server:

```
PUT /hello.html HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.6 [en] (WinNT; I)
Pragma: no-cache
Host: www.ibiblio.org
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
Content-Length: 364

<!doctype html public "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
   <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
   <meta name="Author" content="Elliotte Rusty Harold">
   <meta name="GENERATOR" content="Mozilla/4.6 [en] (WinNT; I) [Netscape]">
   <title>Mine</title>
</head>
<body>
<b>Hello</b>
</body>
</html>
```

As with deleting files, allowing arbitrary users to PUT files on your web server is a clear security risk. Generally, some sort of authentication is required and the server must be specially configured to support PUT. The details are likely to vary from server to server. Most web servers do not include full support for PUT out of the box. For instance, Apache requires you to install an additional module just to handle PUT requests.

15.11.1.5. TRACE

The TRACE request method sends the HTTP header that the server received from the client. The main reason for this information is to see what any proxy servers between the server and client might be changing. For example, suppose this TRACE request is sent:

```
TRACE /xml/ HTTP/1.1
Hello: Push me
User-Agent: Java/1.4.2_05
Host: www.ibiblio.org
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: close
```

The server should respond like this:

```
Date: Thu, 19 Aug 2004 17:50:02 GMT
Server: Apache/1.3.4 (Unix) PHP/3.0.6 mod_perl/1.17
Connection: close
Transfer-Encoding: chunked
Content-Type: message/http
content-length: 169

TRACE /xml/ HTTP/1.1
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: close
Hello: Push me
Host: www.ibiblio.org
User-Agent: Java/1.4.2_05
```

The first six lines are the server's normal response HTTP header. The lines from `TRACE /xml/ HTTP/1.1` on are the echo of the original client request. In this case, the echo is faithful, although out of order. However, if there were a proxy server between the client and server, it might not be.

15.11.2. Disconnecting from the Server

Recent versions of HTTP support what's known as *Keep-Alive*. Keep-Alive enhances the performance of some web connections by allowing multiple requests and responses to be sent in a series over a single TCP connection. A client indicates that it's willing to use HTTP Keep-Alive by including a Connection field in the HTTP request header with the value Keep-Alive:

```
Connection: Keep-Alive
```

However, when Keep-Alive is used, the server can no longer close the connection simply because it has sent the last byte of data to the client. The client may, after all, send another request. Consequently, it is up to the client to close the connection when it's done.

Java marginally supports HTTP Keep-Alive, mostly by piggybacking on top of browser support. It doesn't provide any convenient API for making multiple requests over the same connection. However, in anticipation of a day when Java will better support Keep-Alive, the `HttpURLConnection` class adds a `disconnect( )` method that allows the client to break the connection:

```
public abstract void disconnect( )
```

In practice, you rarely if ever need to call this.

15.11.3. Handling Server Responses

The first line of an HTTP server's response includes a numeric code and a message indicating what sort of response is made. For instance, the most common response is 200 OK, indicating that the requested document was found. For example:

```
HTTP/1.1 200 OK
Date: Fri, 20 Aug 2004 15:33:40 GMT
Server: Apache/1.3.4 (Unix) PHP/3.0.6 mod_perl/1.17
Last-Modified: Sun, 06 Jun 1999 16:30:33 GMT
ETag: "28d907-657-375aa229"
Accept-Ranges: bytes
Content-Length: 1623
Connection: close
Content-Type: text/html

<HTML>
<HEAD>
rest of document follows...
```

Another response that you're undoubtedly all too familiar with is 404 Not Found, indicating that the URL you requested no longer points to a document. For example:

```
HTTP/1.1 404 Not Found
Date: Fri, 20 Aug 2004 15:39:16 GMT
Server: Apache/1.3.4 (Unix) PHP/3.0.6 mod_perl/1.17
Last-Modified: Mon, 20 Sep 1999 19:25:05 GMT
ETag: "5-14ab-37e68a11"
Accept-Ranges: bytes
Content-Length: 5291
Connection: close
Content-Type: text/html

<html>
<head>
<title>Lost ... and lost</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
</head>

<body bgcolor="#FFFFFF">
<div align="left">
  <h1>404 FILE NOT FOUND</h1>
Rest of error message follows...
```

There are many other, less common responses. For instance, code 301 indicates that the resource has permanently moved to a new location and the browser should redirect itself to the new location and update any bookmarks that point to the old location. For example:

```
HTTP/1.1 301 Moved Permanently
Date: Fri, 20 Aug 2004 15:36:44 GMT
Server: Apache/1.3.4 (Unix) PHP/3.0.6 mod_perl/1.17
Location: http://www.ibiblio.org/javafaq/books/beans/index.html
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
Connection: close
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML><HEAD>
<TITLE>301 Moved Permanently</TITLE>
</HEAD><BODY>
<H1>Moved Permanently</H1>
The document has moved <A HREF="http://www.ibiblio.org/javafaq/books/beans/index
.html">here</A>.<P>
<HR>
<ADDRESS>Apache/1.3.4 Server at www.ibiblio.org Port 80</ADDRESS>
</BODY></HTML>
```

The first line of this response is called the *response message*. It will not be returned by the various `getHeaderField( )` methods in `URLConnection`. However, `HttpURLConnection` has a method to read and return just the response message. This is the aptly named `getResponseMessage()`:

```
public String getResponseMessage( ) throws IOException
```

Often all you need from the response message is the numeric response code. `HttpURLConnection` also has a `getResponseCode( )` method to return this as an `int`:

```
public int getResponseCode( ) throws IOException
```

HTTP 1.0 defines 16 response codes. HTTP 1.1 expands this to 40 different codes. While some numbers, notably 404, have become slang almost synonymous with their semantic meaning, most of them are less familiar. The `HttpURLConnection` class includes 36 named constants representing the most common response codes. These are summarized in Table 15-3.

Table 15-3. The HTTP 1.1 response codes

Code	Meaning	HttpURLConnection constant
1XX	Informational	
100	The server is prepared to accept the request body and the client should send it; a new feature in HTTP 1.1 that allows clients to ask whether the server will accept a request before they send a large amount of data as part of the request.	N/A
101	The server accepts the client's request in the Upgrade header field to change the application protocol; e.g., from HTTP 1.0 to HTTP 1.1.	N/A

Code	Meaning	HttpURLConnection constant
2XX	Request succeeded.	
200	The most common response code. If the request method was GET or POST, the requested data is contained in the response along with the usual headers. If the request method was HEAD, only the header information is included.	`HTTP_OK`
201	The server has created a resource at the URL specified in the body of the response. The client should now attempt to load that URL. This code is sent only in response to POST requests.	`HTTP_CREATED`
202	This rather uncommon response indicates that a request (generally from POST) is being processed, but the processing is not yet complete, so no response can be returned. However, the server should return an HTML page that explains the situation to the user and provide an estimate of when the request is likely to be completed, and, ideally, a link to a status monitor of some kind.	`HTTP_ACCEPTED`
203	The resource representation was returned from a caching proxy or other local source and is not guaranteed to be up to date.	`HTTP_NOT_AUTHORITATIVE`
204	The server has successfully processed the request but has no information to send back to the client. This is normally the result of a poorly written form-processing program on the server that accepts data but does not return a response to the user.	`HTTP_NO_CONTENT`
205	The server has successfully processed the request but has no information to send back to the client. Furthermore, the client should clear the form to which the request is sent.	`HTTP_RESET`
206	The server has returned the part of the document the client requested using the byte range extension to HTTP, rather than the whole document.	`HTTP_PARTIAL`
3XX	Relocation and redirection.	

Code	Meaning	HttpURLConnection constant
300	The server is providing a list of different representations (e.g., PostScript and PDF) for the requested document.	HTTP_MULT_CHOICE
301	The resource has moved to a new URL. The client should automatically load the resource at this URL and update any bookmarks that point to the old URL.	HTTP_MOVED_PERM
302	The resource is at a new URL temporarily, but its location will change again in the foreseeable future; therefore, bookmarks should not be updated.	HTTP_MOVED_TEMP
303	Generally used in response to a POST form request, this code indicates that the user should retrieve a document other than the one requested (as opposed to a different location for the requested document).	HTTP_SEE_OTHER
304	The If-Modified-Since header indicates that the client wants the document only if it has been recently updated. This status code is returned if the document has not been updated. In this case, the client should load the document from its cache.	HTTP_NOT_MODIFIED
305	The Location header field contains the address of a proxy that will serve the response.	HTTP_USE_PROXY
307	Almost the same as code 303, a 307 response indicates that the resource has moved to a new URL, although it may move again to a different URL in the future. The client should automatically load the page at this URL.	N/A
4XX	Client error.	
400	The client request to the server used improper syntax. This is rather unusual in normal web browsing but more common when debugging custom clients.	HTTP_BAD_REQUEST
401	Authorization, generally a username and password, is required to access this page. Either a username and password have not yet been presented or the username and password are invalid.	HTTP_UNAUTHORIZED

Code	Meaning	HttpURLConnection constant
402	Not used today, but may be used in the future to indicate that some sort of digital cash transaction is required to access the resource.	HTTP_PAYMENT_REQUIRED
403	The server understood the request, but is deliberately refusing to process it. Authorization will not help. This might be used when access to a certain page is denied to a certain range of IP addresses.	HTTP_FORBIDDEN
404	This most common error response indicates that the server cannot find the requested resource. It may indicate a bad link, a document that has moved with no forwarding address, a mistyped URL, or something similar.	HTTP_NOT_FOUND
405	The request method is not allowed for the specified resource; for instance, you tried to PUT a file on a web server that doesn't support PUT or tried to POST to a URI that only allows GET.	HTTP_BAD_METHOD
406	The requested resource cannot be provided in a format the client is willing to accept, as indicated by the Accept field of the request HTTP header.	HTTP_NOT_ACCEPTABLE
407	An intermediate proxy server requires authentication from the client, probably in the form of a username and password, before it will retrieve the requested resource.	HTTP_PROXY_AUTH
408	The client took too long to send the request, perhaps because of network congestion.	HTTP_CLIENT_TIMEOUT
409	A temporary conflict prevents the request from being fulfilled; for instance, two clients are trying to PUT the same file at the same time.	HTTP_CONFLICT
410	Like a 404, but makes a stronger assertion about the existence of the resource. The resource has been deliberately deleted (not moved) and will not be restored. Links to it should be removed.	HTTP_GONE

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Code	Meaning	HttpURLConnection constant
411	The client must but did not send a Content-length field in the client request HTTP header.	HTTP_LENGTH_REQUIRED
412	A condition for the request that the client specified in the request HTTP header is not satisfied.	HTTP_PRECON_FAILED
413	The body of the client request is larger than the server is able to process at this time.	HTTP_ENTITY_TOO_LARGE
414	The URI of the request is too long. This is important to prevent certain buffer overflow attacks.	HTTP_REQ_TOO_LONG
415	The server does not understand or accept the MIME content-type of the request body.	HTTP_UNSUPPORTED_TYPE
416	The server cannot send the byte range the client requested.	N/A
417	The server cannot meet the client's expectation given in an Expect-request header field.	N/A
5XX	Server error.	
500	An unexpected condition occurred that the server does not know how to handle.	HTTP_SERVER_ERROR HTTP_INTERNAL_ERROR
501	The server does not have a feature that is needed to fulfill this request. A server that cannot handle POST requests might send this response to a client that tried to POST form data to it.	HTTP_NOT_IMPLEMENTED
502	This code is applicable only to servers that act as proxies or gateways. It indicates that the proxy received an invalid response from a server it was connecting to in an effort to fulfill the request.	HTTP_BAD_GATEWAY
503	The server is temporarily unable to handle the request, perhaps due to overloading or maintenance.	HTTP_UNAVAILABLE

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Code	Meaning	HttpURLConnection constant
504	The proxy server did not receive a response from the upstream server within a reasonable amount of time, so it can't send the desired response to the client.	HTTP_GATEWAY_TIMEOUT
505	The server does not support the version of HTTP the client is using (e.g., the as-yet-nonexistent HTTP 2.0).	HTTP_VERSION

Example 15-10 is a revised source viewer program that now includes the response message. The lines added since `SourceViewer2` are in bold.

Example 15-10. A SourceViewer that includes the response code and message

```
import java.net.*;
import java.io.*;
import javax.swing.*;
import java.awt.*;

public class SourceViewer3 {

  public static void main (String[] args) {

    for (int i = 0; i < args.length; i++) {
      try {

        //Open the URLConnection for reading
        URL u = new URL(args[i]);
        HttpURLConnection uc = (HttpURLConnection) u.openConnection( );
        int code = uc.getResponseCode( );
        String response = uc.getResponseMessage( );
        System.out.println("HTTP/1.x " + code + " " + response);
        for (int j = 1; ; j++) {
          String header = uc.getHeaderField(j);
          String key = uc.getHeaderFieldKey(j);
          if (header == null || key == null) break;
          System.out.println(uc.getHeaderFieldKey(j) + ": " + header);
        }  // end for
        InputStream in = new BufferedInputStream(uc.getInputStream( ));
        // chain the InputStream to a Reader
        Reader r = new InputStreamReader(in);
        int c;
        while ((c = r.read( )) != -1) {
          System.out.print((char) c);
        }
      }
      catch (MalformedURLException ex) {
        System.err.println(args[0] + " is not a parseable URL");
      }
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
          catch (IOException ex) {
            System.err.println(ex);
          }

        } //  end if

      } // end main

    }  // end SourceViewer3
```

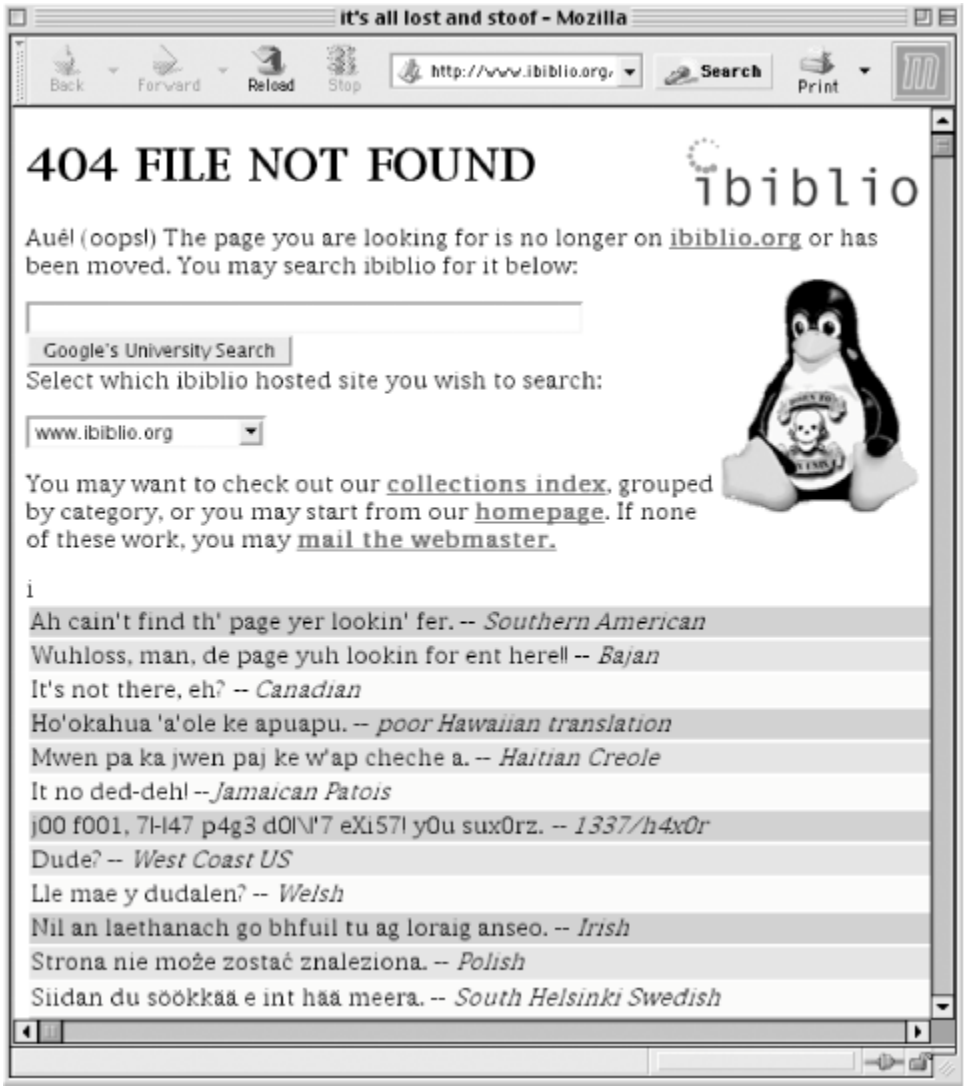
The only thing this program doesn't read that the server sends is the version of HTTP the server is using. There's currently no method to return that. If you need it, you'll just have to use a raw socket instead. Consequently, in this example, we just fake it as "HTTP/1.x", like this:

```
% java SourceViewer3 http://www.oreilly.com
HTTP/1.x 200 OK
Server: WN/1.15.1
Date: Mon, 01 Nov 1999 23:39:19 GMT
Last-modified: Fri, 29 Oct 1999 23:40:06 GMT
Content-type: text/html
Title: www.oreilly.com -- Welcome to O'Reilly &amp;amp; Associates! --
computer  books, software, online publishing
Link: <mailto:webmaster@ora.com>; rev="Made"
<HTML>
<HEAD>
...
```

15.11.3.1. Error conditions

On occasion, the server encounters an error but returns useful information in the message body nonetheless. For example, when a client requests a nonexistent page from the *www.ibiblio.org* web site, rather than simply returning a 404 error code, the server sends the search page shown in Figure 15-2 to help the user figure out where the missing page might have gone.

Figure 15-2. IBiblio's 404 page



The `getErrorStream( )` method returns an `InputStream` containing this data or `null` if no error was encountered or no data returned:

```
public InputStream getErrorStream( ) // Java 1.2
```

In practice, this isn't necessary. Most implementations will return this data from `getInputStream()` as well.

15.11.3.2. Redirects

The 300-level response codes all indicate some sort of redirect; that is, the requested resource is no longer available at the expected location but it may be found at some other location.

When encountering such a response, most browsers automatically load the document from its new location. However, this can be a security risk, because it has the potential to move the user from a trusted site to an untrusted one, perhaps without the user even noticing.

By default, an `HttpURLConnection` follows redirects. However, the `HttpURLConnection` class has two static methods that let you decide whether to follow redirects:

```
public static boolean getFollowRedirects( )
public static void    setFollowRedirects(boolean follow)
```

The `getFollowRedirects( )` method returns `true` if redirects are being followed, `false` if they aren't. With an argument of true, the `setFollowRedirects( )` method makes `HttpURLConnection` objects follow redirects. With an argument of `false`, it prevents them from following redirects. Since these are static methods, they change the behavior of all `HttpURLConnection` objects constructed after the method is invoked. The `setFollowRedirects( )` method may throw a `SecurityException` if the security manager disallows the change. Applets especially are not allowed to change this value.

Java has two methods to configure redirection on an instance-by-instance basis. These are:

```
public boolean getInstanceFollowRedirects( )   // Java 1.3
public void    setInstanceFollowRedirects(boolean followRedirects)  // Java 1.3
```

If `setInstanceFollowRedirects( )` is not invoked on a given `HttpURLConnection`, that `HttpURLConnection` simply follows the default behavior as set by the class method `HttpURLConnection.setFollowRedirects( )`.

15.11.4. Proxies

Many users behind firewalls or using AOL or other high-volume ISPs access the web through proxy servers. The `usingProxy( )` method tells you whether the particular `HttpURLConnection` is going through a proxy server:

```
public abstract boolean usingProxy( ) // Java 1.3
```

It returns `true` if a proxy is being used, `false` if not. In some contexts, the use of a proxy server may have security implications.

15.11.5. Streaming Mode

Every request sent to an HTTP server has an HTTP header. One field in this header is the Content-length; that is, the number of bytes in the body of the request. The header comes before the body. However, to write the header you need to know the length of the body, which you may not have yet. Normally the way Java solves this Catch-22 is by caching every thing you write onto the `OutputStream` retrieved from the `HttpURLConnection` until the stream is closed. At that point, it knows how many bytes are in the body so it has enough information to write the Content-length header.

This scheme is fine for small requests sent in response to typical web forms. However, it's burdensome for responses to very long forms or some SOAP messages. It's very wasteful and slow for medium-to-large documents sent with HTTP PUT. It's much more efficient if Java doesn't have to wait for the last byte of data to be written before sending the first byte of data over the network. Java 1.5 offers two solutions to this problem. If you know the size of your data—for instance, you're uploading a file of known size using HTTP PUT—you can tell the `HttpURLConnection` object the size of that data. If you don't know the size of the data in advance, the you can use chunked transfer encoding instead. In chunked transfer encoding, the body of the request is sent in multiple pieces, each with its own separate content length. To turn on chunked transfer encoding, just pass the size of the chunks you want to the `setChunkedStreamingMode( )` method before you connect the URL.

```
public void setChunkedStreamingMode(int chunkLength) // Java 1.5
```

Java will then use a slightly different form of HTTP than the examples in this book. However, to the Java programmer the difference is irrelevant. As long as you're using the `URLConnection` class instead of raw sockets and as long as the server supports chunked transfer encoding, it should all just work without any further changes to your code. However, not all servers support chunked encoding, though most of the late-model, major ones do. Even more importantly, chunked transfer encoding does get in the way of authentication and redirection. If you're trying to send chunked files to a redirected URL or one that requires password authentication, an `HttpRetryException` will be thrown. You'll then need to retry the request at the new URL or at the old URL with the appropriate credentials; and this all needs to be done manually without the full support of the HTTP protocol handler you normally have. Therefore, don't use chunked transfer encoding unless you really need it. As with most performance advice, this means you shouldn't implement this optimization until measurements prove the non-streaming default is a bottleneck.

If you do happen to know the size of the request data in advance, Java 1.5 lets you optimize the connection by providing this information to the `HttpURLConnection` object. If you do this Java can start streaming the data over the network immediately. Otherwise, it has to

cache everything you write in order to determine the content length, and only send it over the network after you've closed the stream. If you know exactly how big your data is, pass that number to the `setFixedLengthStreamingMode( )` method:

```
public void setFixedLengthStreamingMode(int contentLength)
```

Java will use this number in the HTTP Content-length HTTP header field. However, if you then try to write more or less than the number of bytes given here, Java will throw an `IOException`. Of course, that will happen later, when you're writing data, not when you first call this method. The `setFixedLengthStreamingMode( )` method itself will throw an `IllegalArgumentException` if you pass in a negative number, or an `IllegalStateException` if the connection is connected or has already been set to chunked transfer encoding. (You can't use both chunked transfer encoding and fixed-length streaming mode on the same request.)

Fixed-length streaming mode is transparent on the server side. Servers neither know nor care how the Content-length was set as long as it's correct. However, like chunked transfer encoding, streaming mode does interfere authentication and redirection. If either of these is required for a given URL, an `HttpRetryException` will be thrown; you have to manually retry. Therefore, don't use this mode unless you really need it.

15.12. Caches

Web browsers have been caching pages and images for years. If a logo is repeated on every page of a site, the browser normally loads it from the remote server only once, stores it in its cache, and reloads it from the cache whenever it's needed rather than returning to the remote server every time the same page is needed. Several HTTP headers, including Expires and Cache-Control, can control caching.

Java 1.5 finally adds the ability to cache data to the `URL` and `URLConnection` classes. By default, Java 1.5 does not cache anything, but you can create your own cache by subclassing the `java.net.ResponseCache` class and installing it as the system default. Whenever the system tries to load a new URL thorough a protocol handler, it will first look for it in the cache. If the cache returns the desired content, the protocol handler won't need to connect to the remote server. However, if the requested data is not in the cache, the protocol handler will download it. After it's done so, it will put its response into the cache so the content is more quickly available the next time that URL is loaded.

Two abstract methods in the `ResponseCache` class store and retrieve data from the system's single cache:

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
public abstract CacheResponse get(URI uri, String requestMethod,
 Map<String,List<String>> requestHeaders) throws IOException
public abstract CacheRequest put(URI uri, URLConnection connection)
 throws IOException
```

The `put( )` method returns a `CacheRequest` object that wraps an `OutputStream` into which the protocol handler will write the data it reads. `CacheRequest` is an abstract class with two methods, as shown in Example 15-11.

Example 15-11. The CacheRequest class

```
package java.net

public abstract class CacheRequest {

  public abstract OutputStream getBody( ) throws IOException;
  public abstract void abort( );

}
```

The `getOutputStream()` method in the subclass should return an `OutputStream` that points into the cache's data store for the URI passed to the `put( )` method at the same time. For instance, if you're storing the data in a file, then you'd return a `FileOutputStream` connected to that file. The protocol handler will copy the data it reads onto this `OutputStream`. If a problem arises while copying (e.g., the server unexpectedly closes the connection), the protocol handler calls the `abort( )` method. This method should then remove any data that has been stored from the cache.

Example 15-12 demonstrates a basic `CacheRequest` subclass that passes back a `ByteArrayOutputStream`. Later the data can be retrieved using the `getData( )` method, a custom method in this subclass just retrieving the data Java wrote onto the `OutputStream` this class supplied. An obvious alternative strategy would be to store results in files and use a `FileOutputStream` instead.

Example 15-12. A basic CacheRequest subclass

```
import java.net.*;
import java.io.*;
import java.util.*;

public class SimpleCacheRequest extends CacheRequest {
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
    ByteArrayOutputStream out = new ByteArrayOutputStream( );

    public OutputStream getBody( ) throws IOException {
      return out;
    }

    public void abort( ) {
      out = null;
    }

    public byte[] getData( ) {
      if (out == null) return null;
      else return out.toByteArray( );
    }

  }
```

The `get( )` method retrieves the data and headers from the cache and returns them wrapped in a `CacheResponse` object. It returns `null` if the desired URI is not in the cache, in which case the protocol handler loads the URI from the remote server as normal. Again, this is an abstract class that you have to implement in a subclass. Example 15-13 summarizes this class. It has two methods, one to return the data of the request and one to return the headers. When caching the original response, you need to store both. The headers should be returned in an unmodifiable map with keys that are the HTTP header field names and values that are lists of values for each named HTTP header.

Example 15-13. The CacheResponse class

```
package java.net;

public abstract class CacheRequest {

  public abstract InputStream getBody( ) ;
  public abstract Map<String,List<String>> getHeaders( );

}
```

Example 15-14 shows a simple `CacheResponse` subclass that is tied to a `SimpleCacheRequest`. In this example, shared references pass data from the request class to the response class. If we were storing responses in files, we'd just need to share the filenames instead. Along with the `SimpleCacheRequest` object from which it will read the data, we must also pass the original `URLConnection` object into the constructor. This is used to read the HTTP header so it can be stored for later retrieval. The object also keeps track of the expiration date (if any) provided by the server for the cached representation of the resource.

Example 15-14. A basic CacheResponse subclass

```
import java.net.*;
import java.io.*;
import java.util.*;

public class SimpleCacheResponse extends CacheResponse {

  private Map<String,List<String>> headers;
  private SimpleCacheRequest request;
  private Date expires;

  public SimpleCacheResponse(SimpleCacheRequest request, URLConnection uc)
   throws IOException {

    this.request = request;

    // deliberate shadowing; we need to fill the map and
    // then make it unmodifiable
    Map<String,List<String>> headers = new HashMap<String,List<String>>( );
    String value = "";
    for (int i = 0;; i++) {
       String name = uc.getHeaderFieldKey(i);
       value = uc.getHeaderField(i);
       if (value == null) break;
       List<String> values = headers.get(name);
       if (values == null) {
         values = new ArrayList<String>(1);
         headers.put(name, values);
       }
       values.add(value);
    }
    long expiration = uc.getExpiration( );
    if (expiration != 0) {
      this.expires = new Date(expiration);
    }

    this.headers = Collections.unmodifiableMap(headers);

  }

  public InputStream getBody( ) {
    return new ByteArrayInputStream(request.getData( ));
  }

  public Map<String,List<String>> getHeaders( )
   throws IOException {
    return headers;
  }

  public boolean isExpired( ) {
    if (expires == null) return false;
    else {
      Date now = new Date( );
      return expires.before(now);
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
        }
      }

    }
```

Finally, we need a simple `ResponseCache` subclass that passes `SimpleCacheRequest`s and `SimpleCacheResponse`s back to the protocol handler as requested. Example 15-15 demonstrates such a simple class that stores a finite number of responses in memory in one big `HashMap`.

Example 15-15. An in-memory ResponseCache

```
import java.net.*;
import java.io.*;
import java.util.*;
import java.util.concurrent.*;

public class MemoryCache extends ResponseCache {

  private Map<URI, SimpleCacheResponse> responses
   = new ConcurrentHashMap<URI, SimpleCacheResponse>( );
  private int maxEntries = 100;

  public MemoryCache( ) {
    this(100);
  }

  public MemoryCache(int maxEntries) {
    this.maxEntries = maxEntries;
  }

  public CacheRequest put(URI uri, URLConnection uc)
   throws IOException {

     if (responses.size( ) >= maxEntries) return null;

     String cacheControl = uc.getHeaderField("Cache-Control");
     if (cacheControl != null && cacheControl.indexOf("no-cache") >= 0) {
       return null;
     }

     SimpleCacheRequest request = new SimpleCacheRequest( );
     SimpleCacheResponse response = new SimpleCacheResponse(request, uc);

     responses.put(uri, response);
     return request;

  }

  public CacheResponse get(URI uri, String requestMethod,
   Map<String,List<String>> requestHeaders)
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
        throws IOException {

          SimpleCacheResponse response = responses.get(uri);
          // check expiration date
          if (response != null && response.isExpired( )) {
            responses.remove(response);
            response = null;
          }
          return response;

        }

      }
```

Once a `ResponseCache` like this one is installed, Java's HTTP protocol handler always uses it, even when it shouldn't. The client code needs to check the expiration dates on anything it's stored and watch out for Cache-Control header fields. The key value of concern is no-cache. If you see this string in a Cache-Control header field, it means any resource representation is valid only momentarily and any cached copy is likely to be out of date almost immediately, so you really shouldn't store it at all.

Each retrieved resource stays in the `HashMap` until it expires. This example waits for an expired document to be requested again before it deletes it from the cache. A more sophisticated implementation could use a low-priority thread to scan for expired documents and remove them to make way for others. Instead of or in addition to this, an implementation might cache the representations in a queue and remove the oldest documents or those closest to their expiration date as necessary to make room for new ones. An even more sophisticated implementation could track how often each document in the store was accessed and expunge only the oldest and least-used documents.

I've already mentioned that you could implement this on top of the filesystem instead of sitting on top of the Java Collections API. You could also store the cache in a database and you could do a lot of less-common things as well. For instance, you could redirect requests for certain URLs to a local server rather than a remote server halfway around the world, in essence using a local web server as the cache. Or a `ResponseCache` could load a fixed set of files at launch time and then only serve those out of memory. This might be useful for a server that processes many different SOAP requests, all of which adhere to a few common schemas that can be stored in the cache. The abstract `ResponseCache` class is flexible enough to support all of these and other usage patterns.

Regrettably, Java only allows one cache at a time. To change the cache object, use the static `ResponseCache.setDefault()` and `ResponseCache.getDefault( )` methods:

```
public static ResponseCache getDefault( )
public static void setDefault(ResponseCache responseCache)
```

These set the single cache used by all programs running within the same Java virtual machine. For example, this one line of code installs Example 15-13 in an application:

```
ResponseCache.setDefault(new MemoryCache( ));
```

15.13. JarURLConnection

Applets often store their *.class* files in a JAR archive, which bundles all the classes in one package that still maintains the directory hierarchy needed to resolve fully qualified class names like `com.macfaq.net.QueryString`. Furthermore, since the entire archive is compressed and can be downloaded in a single HTTP connection, it requires much less time to download the *.jar* file than to download its contents one file at a time. Some programs store needed resources such as sounds, images, and even text files inside these JAR archives. Java provides several mechanisms for getting the resources out of the JAR archive, but the one that we'll address here is the *jar* URL. The `JarURLConnection` class supports URLs that point inside JAR archives:

```
public abstract class JarURLConnection extends URLConnection// Java 1.2
```

A *jar* URL starts with a normal URL that points to a JAR archive, such as http://www.cafeaulait.org/network.jar or *file:///D%7C/javafaq/network.jar*. Then the protocol *jar*: is prefixed to this URL. Finally, *!/* and the path to the desired file inside the JAR archive are suffixed to the original URL. For example, to find the file *com/macfaq/net/QueryString.class* inside the previous *.jar* files, you'd use the URLs *jar:http://www.cafeaulait.org/network.jar!/com/macfaq/net/QueryString.class* or *jar:file://D%7C/javafaq/network.jar!/com/macfaq/net/QueryString.class*. Of course, this isn't limited simply to Java *.class* files. You can use *jar* URLs to point to any kind of file that happens to be stored inside a JAR archive, including images, sounds, text, HTML files, and more. If the path is left off, the URL refers to the entire JAR archive, e.g., *jar:http://www.cafeaulait.org/network.jar!/* or *jar:file:///D%7C/javafaq/network.jar!/*.

Web browsers don't understand *jar* URLs, though. They're used only inside Java programs. To get a `JarURLConnection`, construct a `URL` object using a *jar* URL and cast the return value of its `openConnection()` method to `JarURLConnection`. Java downloads the entire JAR archive to a temporary file, opens it, and positions the file pointer at the beginning of the particular entry you requested. You can then read the contents of the particular file inside the JAR archive using the `InputStream` returned by `getInputStream( )`. For example:

```
try {
  //Open the URLConnection for reading
  URL u = new URL(
   "jar:http://www.cafeaulait.org/course/week1.jar!/week1/05.html");
  URLConnection uc = u.openConnection( );

  InputStream in = uc.getInputStream( );
  // chain the InputStream to a Reader
  Reader r = new InputStreamReader(in);
  int c;
  while ((c = r.read( )) != -1) {
    System.out.print((char) c);
  }
}
catch (IOException ex) {
  System.err.println(ex);
}
```

Besides the usual methods of the `URLConnection` class that `JarURLConnection` inherits, this class adds eight new methods, mostly to return information about the JAR archive itself. These are:

```
public URL           getJarFileURL( )                        // Java 1.2
public String        getEntryName( )                         // Java 1.2
public JarEntry      getJarEntry( ) throws IOException       // Java 1.2
public Manifest      getManifest( ) throws IOException       // Java 1.2
public Attributes    getAttributes( ) throws IOException     // Java 1.2
public Attributes    getMainAttributes( ) throws IOException // Java 1.2
public Certificate[] getCertificates( ) throws IOException   // Java 1.2
public abstract JarFile getJarFile( ) throws IOException     // Java 1.2
```

The `getJarFileURL()` method is the simplest. It merely returns the URL of the *jar* file being used by this connection. This generally differs from the URL of the file in the archive being used for this connection. For instance, the *jar* file URL of *jar:http://www.cafeaulait.org/network.jar!/com/macfaq/net/QueryString.class* is *http://www.cafeaulait.org/network.jar*. The `getEntryName( )` returns the other part of the *jar* URL; that is, the path to the file inside the archive. The entry name of *jar:http://www.cafeaulait.org/network.jar!/com/macfaq/net/QueryString.class* is *com/macfaq/net/QueryString.class*.

The `getJarFile( )` method returns a `java.util.jar.JarFile` object that you can use to inspect and manipulate the archive contents. The `getJarEntry( )` method returns a `java.util.jar.JarEntry` object for the particular file in the archive that this `URLConnection` is connected to. It returns null if the URL points to a whole JAR archive rather than a particular entry in the archive.

Much of the functionality of both `JarFile` and `JarEntry` is duplicated by other methods in the `JarURLConnection` class; which to use is mostly a matter of personal preference. For instance, the `getManifest( )` method returns a `java.util.jar.Manifest` object representing the contents of the JAR archive's manifest file. A *manifest file* is included in the archive to supply metainformation about the contents of the archive, such as which file

contains the `main( )` method and which classes are Java beans. It's called *MANIFEST.MF* and placed in the *META-INF* directory; its contents typically look something like this:

```
Manifest-Version: 1.0
Required-Version: 1.0

Name: com/macfaq/net/FormPoster.class
Java-Bean: true
Last-modified: 10-21-2003
Depends-On: com/macfaq/net/QueryString.class
Digest-Algorithms: MD5
MD5-Digest: XD4578YEEIK9MGX54RFGT7UJUI9810

Name: com/macfaq/net/QueryString.class
Java-Bean: false
Last-modified: 5-17-2003
Digest-Algorithms: MD5
MD5-Digest: YP7659YEEIK0MGJ53RYHG787YI8900
```

The name-value pairs associated with each entry are called the *attributes* of that entry. The name-value pairs not associated with any entry are called the *main attributes* of the archive. The `getAttributes()` method returns a `java.util.jar.Attributes` object representing the attributes that the manifest file specifies for this *jar* entry, or `null` if the URL points to a whole JAR archive. The `getMainAttributes( )` method returns a `java.util.jar.Attributes` object representing the attributes that the manifest file specifies for the entire JAR archive as a whole.

Finally, the `getCertificates()` method returns an array of digital signatures (each represented as a `java.security.cert.Certificate` object) that apply to this *jar* entry, or `null` if the URL points to a JAR archive instead of a particular entry. These are actually read from separate signature files for each *jar* entry, not from the manifest file. Unlike the other methods of `JarURLConnection`, `getCertificates( )` can be called only after the entire input stream for the *jar* URL has been read. This is because the current hash of the data needs to be calculated, which can be done only when the entire entry is available.

More details about the `java.util.jar` package, JAR archives, manifest files, entries, attributes, digital signatures, how this all relates to Zip files and Zip and JAR streams, and so forth can be found on Sun's web site at http://java.sun.com/j2se/1.4.2/docs/guide/jar/ or in Chapter 9 of my book, *Java I/O* (O'Reilly).

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004