CHAPTER 5
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. A `URLConnection` can inspect the header sent by the server and respond accordingly. It can set the header fields used in the client request. Finally, a `URLConnection` can send data back to a web server with `POST`, `PUT`, and other HTTP request methods.

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 *com- press*, 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.

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 sub- classes. 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.

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, you can skip step 3. If you only want the data from the server and don't care about any meta-information, or if the protocol doesn't provide any meta-information, you can 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)
```

Consequently, unless you're subclassing `URLConnection` to handle a new kind of URL (i.e., writing a protocol handler), you create one of these objects by invoking the `openConnection()` method of the `URL` class. For example:

```
try {
  URL u = new URL("http://www.overcomingbias.com/");
  URLConnection uc = u.openConnection();
  // read from the URL...
} catch (MalformedURLException ex) {
  System.err.println(ex);
} catch (IOException ex) {
  System.err.println(ex);
```

```
}
```

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 serv- er:

```
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()`, `getHeader Field()`, 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.

Reading Data from a Server

The following 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. Example 7-1 uses the `getInputStream()` method to download a web page.

Example 7-1. Download a web page with a URLConnection

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

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();
        try (InputStream raw = uc.getInputStream()) { // autoclose
          InputStream buffer = new BufferedInputStream(raw); // chain the InputStream to a
          Reader
          Reader reader = new InputStreamReader(buffer);
          int c;
          while ((c = reader.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);
      }
    }
  }
}
```

It is no accident that this program is almost the same as Example 5-2. The `open Stream()` 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.

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/1.1 301 Moved Permanently
Date: Sun, 21 Apr 2013 15:12:46 GMT
Server: Apache
Location: http://www.ibiblio.org/
Content-Length: 296
Connection: close
Content-Type: text/html; charset=iso-8859-1
```

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 `URL Connection` 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 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 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.

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

public String getContentType()

The `getContentType()` method returns the MIME media type of the response body. It relies on the web server to send a valid content type. 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, 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: application/xml; charset=iso-2022-jp
```

In this case, `getContentType()` returns the full value of the Content-type field, including the character encoding. You can use this to improve on Example 7-1 by using the en- coding 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 7-2 demonstrates.

Example 7-2. Download a web page with the correct character set

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

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);
        }
        r.close();
      } catch (MalformedURLException ex) {
      System.err.println(args[0] + " is not a parseable URL");
      } catch (UnsupportedEncodingException ex) {
      System.err.println(
           "Server sent an encoding Java does not support: " + ex.getMessage()); }
      catch (IOException ex) {
        System.err.println(ex);
      }
    }
  }
}
```

public int getContentLength()

The `getContentLength()` method tells you how many bytes there are in the content. 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.

As networks get faster and files get bigger, it is actually possible to find resources whose size exceeds the maximum `int` value (about 2.1 billion bytes). In this case, `getConten tLength()` returns –1. Java 7 adds a `getContentLengthLong()` method that works just like `getContentLength()` except that it returns a `long` instead of an `int` and thus can handle much larger resources:

```
public int getContentLengthLong() // Java 7
```

Chapter 5 showed 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 con- nection exactly where the data is finished; therefore, you don't know when to stop read- ing. To download a binary file, it is more reliable to use a `URLConnection`'s `getConten tLength()` method to find the file's length, then read exactly the number of bytes indicated. Example 7-3 is a program that uses this technique to save a binary file on a disk.

Example 7-3. Downloading a binary file from a website and saving it to disk

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

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);
      }

  }

  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.");
    }

    try (
    InputStream raw = uc.getInputStream()) {
    InputStream in = new BufferedInputStream(raw);
      byte[] data = new byte[contentLength];
      int offset = 0;
      while (offset < contentLength) {
         int bytesRead = in.read(data, offset, data.length - offset);
         if (bytesRead == -1)
         break;
         offset += bytesRead;
      }

      if (offset != contentLength) {
        throw new IOException("Only read " + offset
            + " bytes; Expected " + contentLength + " bytes");
      }
      String filename = u.getFile();
      filename = filename.substring(filename.lastIndexOf('/') + 1);
      try (
      FileOutputStream fout = new FileOutputStream(filename)) {
        fout.write(data);
        fout.flush();
      }
    }
  }
}
```

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 checks are both `false`, you have a binary file of known length: that's what you want.

Now that you have a genuine binary file on your hands, you 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. You 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 you're 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 `file`

`name.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)`.
`AutoCloseable` is used to clean up throughout.

public String getContentEncoding()

The `getContentEncoding()` 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`.

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.

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 the document expires. If the HTTP header does not include an Expiration field, `getExpiration()` returns 0, which means that the document does not expire and can remain in the cache indefinitely.

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 7-4 reads URLs from the command line and uses these six methods to print their content type, content length, content encoding, date of last modification, expira- tion date, and current date.

Example 7-4. Return the header

```
import java.io.*;
import java.net.*;
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());
        if (uc.getContentEncoding() != null) {
          System.out.println("Content-encoding: "
              + uc.getContentEncoding());
        }
        if (uc.getDate() != 0) {
          System.out.println("Date: " + new Date(uc.getDate()));
        }
        if (uc.getLastModified() != 0) {
          System.out.println("Last modified: "
              + new Date(uc.getLastModified()));
        }
        if (uc.getExpiration() != 0) {
          System.out.println("Expiration date: "
              + new Date(uc.getExpiration()));
        }
        if (uc.getContentLength() != -1) {
          System.out.println("Content-length: " + uc.getContentLength());
```

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

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

```
% java HeaderViewer http://www.oreilly.com Content-type: text/html;
charset=utf-8 Date: Fri May 31 18:08:09 EDT 2013
Last modified: Fri May 31 17:04:14 EDT 2013
Expiration date: Fri May 31 22:08:09 EDT 2013
Content-length: 83273
```

The content type of the file at *http://www.oreilly.com* is `text/html`. No content encoding was used. The file was sent on Friday, May 31, 2013 at 6:08 P.M., Eastern Daylight Time. It was last modified on the same day at 5:04 P.M. and it expires four hours in the future.

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:// oreilly.com/favicon.ico*. Now the server sends a Content-length header with a value of 2294:

```
% java HeaderViewer http://oreilly.com/favicon.ico Content-type: image/x-icon
Date: Fri May 31 18:16:01 EDT 2013
Last modified: Wed Mar 26 19:14:36 EST 2003
Expiration date: Fri Jun 07 18:16:01 EDT 2013
Content-length: 2294
```

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 preceding 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`.

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 that the preceding 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 nonnull.

public String getHeaderFieldKey(int n)

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

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

public String getHeaderField(int n)

This method returns the value of the n_{th} header field. In HTTP, the starter line containing the request method and path is header field zero and the first actual header is one. Example 7-5 uses this method in conjunction with `getHeaderFieldKey()` to print the entire HTTP header.

Example 7-5. Print the entire HTTP header

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

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

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

```
% java AllHeaders http://www.oreilly.com Date: Sat, 04 May 2013
11:28:26 GMT Server: Apache
Last-Modified: Sat, 04 May 2013 07:35:04 GMT
Accept-Ranges: bytes
Content-Length: 80366
Content-Type: text/html; charset=utf-8
Cache-Control: max-age=14400
Expires: Sat, 04 May 2013 15:28:26 GMT
Vary: Accept-Encoding
Keep-Alive: timeout=3, max=100
Connection: Keep-Alive
```

Besides the headers with named getter methods, this server also provides Server, Accept-Ranges, Cache-control, Vary, Keep-Alive, and Connection headers. Other servers may have different sets of headers.

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 (e.g., 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, `getHeaderField Date()` 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`.

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.

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 requesting it from the remote server every time the logo is encountered. Several HTTP headers, including Expires and Cache-control, can control caching.

By default, the assumption is that a page accessed with `GET` over HTTP can and should be cached. A page accessed with HTTPS or `POST` usually shouldn't be. However, HTTP headers can adjust this:

- An Expires header (primarily for HTTP 1.0) indicates that it's OK to cache this representation until the specified time.
- The Cache-control header (HTTP 1.1) offers fine-grained cache policies:
  - `max-age=[`*`seconds`*`]`: Number of seconds from now before the cached entry should expire
  - `s-maxage=[`*`seconds`*`]`: Number of seconds from now before the cached entry should expire from a shared cache. Private caches can store the entry for longer.
  - `public`: OK to cache an authenticated response. Otherwise authenticated re- sponses are not cached.
  - `private`: Only single user caches should store the response; shared caches should not.
  - `no-cache`: Not quite what it sounds like. The entry may still be cached, but the client should reverify the state of the resource with an ETag or Last-modified header on each access.
  - `no-store`: Do not cache the entry no matter what.

  Cache-control overrides Expires if both are present. A server can send multiple Cache-control headers in a single header as long as they don't conflict.
- The Last-modified header is the date when the resource was last changed. A client can use a `HEAD` request to check this and only come back for a full `GET` if its local cached copy is older than the Last-modified date.
- The ETag header (HTTP 1.1) is a unique identifier for the resource that changes when the resource does. A client can use a `HEAD` request to check this and only come back for a full `GET` if its local cached copy has a different ETag.

For example, this HTTP response says that the resource may be cached for 604,800 seconds (HTTP 1.1) or one week later (HTTP 1.0). It also says it was last modified on April 20 and has an ETag, so if the local cache already has a copy more recent than that, there's no need to load the whole document now:

```
HTTP/1.1 200 OK
Date: Sun, 21 Apr 2013 15:12:46 GMT
Server: Apache
Connection: close
Content-Type: text/html; charset=ISO-8859-1
Cache-control: max-age=604800
Expires: Sun, 28 Apr 2013 15:12:46 GMT
Last-modified: Sat, 20 Apr 2013 09:55:04 GMT
ETag: "67099097696afcf1b67e"
```

Example 7-6 is a simple Java class for parsing and querying Cache-control headers.

Example 7-6. How to inspect a Cache-control header

```
import java.util.Date;
import java.util.Locale;


public class CacheControl {

  private Date maxAge = null;
```

```
private Date sMaxAge = null;
private boolean mustRevalidate = false;
private boolean noCache = false;
private boolean noStore = false;
private boolean proxyRevalidate = false;
private boolean publicCache = false;
private boolean privateCache = false;

public CacheControl(String s) {
  if (s == null || !s.contains(":")) {
    return; // default policy
  }

  String value = s.split(":")[1].trim();
  String[] components = value.split(",");

  Date now = new Date();
  for (String component : components) {
    try {
      component = component.trim().toLowerCase(Locale.US); if
      (component.startsWith("max-age=")) {
        int secondsInTheFuture = Integer.parseInt(component.substring(8));
        maxAge = new Date(now.getTime() + 1000 * secondsInTheFuture);
      } else if (component.startsWith("s-maxage=")) {
        int secondsInTheFuture = Integer.parseInt(component.substring(8));
        sMaxAge = new Date(now.getTime() + 1000 * secondsInTheFuture);
      } else if (component.equals("must-revalidate")) {
      mustRevalidate = true;
      } else if (component.equals("proxy-revalidate")) {
      proxyRevalidate = true;
      } else if (component.equals("no-cache")) {
      noCache = true;
      } else if (component.equals("public")) {
      publicCache = true;
      } else if (component.equals("private")) {
      privateCache = true;
      }
    } catch (RuntimeException ex) {
    continue;
    }
  }
}

public Date getMaxAge() {
  return maxAge;
}

public Date getSharedMaxAge() {
  return sMaxAge;
}

public boolean mustRevalidate() {
  return mustRevalidate;
}

public boolean proxyRevalidate() {
  return proxyRevalidate;
}

public boolean noStore() {
  return noStore;
}

public boolean noCache() {
  return noCache;
}

public boolean publicCache() {
  return publicCache;
}
```

```
  public boolean privateCache() {
    return privateCache;
  }
}
```

A client can take advantage of this information:

- If a representation of the resource is available in the local cache, and its expiry date has not arrived, just use it. Don't even bother talking to the server.
- If a representation of the resource is available in the local cache, but the expiry date has arrived, check the server with `HEAD` to see if the resource has changed before performing a full `GET`.

Web Cache for Java

By default, Java does not cache anything. To install a system-wide cache of the `URL` class will use, you need the following:

- A concrete subclass of `ResponseCache`
- A concrete subclass of `CacheRequest`
- A concrete subclass of `CacheResponse`

You install your subclass of `ResponseCache` that works with your subclass of `CacheRe quest` and `CacheResponse` by passing it to the static method `ResponseCache.setDe fault()`. This installs your cache object as the system default. A Java virtual machine can only support a single shared cache.

Once a cache is installed whenever the system tries to load a new URL, it will first look for it in the cache. If the cache returns the desired content, the `URLConnection` 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 sys- tem's single cache:

```
        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 `URL` will write cacheable data it reads. `CacheRequest` is an abstract class with two methods, as shown in Example 7-7.

Example 7-7. 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, you'd return a `FileOutput Stream` 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 from the cache that has been stored for this request.

Example 7-8 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 `Output Stream` this class supplied. An obvious alternative strategy would be to store results in files and use a `FileOutputStream` instead.

Example 7-8. A concrete CacheRequest subclass

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

```
public class SimpleCacheRequest extends CacheRequest {

  private ByteArrayOutputStream out = new ByteArrayOutputStream();

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

  @Override
  public void abort() {
    out.reset();
  }

  public byte[] getData() {
    if (out.size() == 0) return null;
    else return out.toByteArray();
  }
}
```

The `get()` method in `ResponseCache` 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 7-9 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 7-9. The CacheResponse class

```
public abstract class CacheResponse {
  public abstract Map<String, List<String>> getHeaders() throws IOException;
  public abstract InputStream getBody() throws IOException;
}
```

Example 7-10 shows a simple `CacheResponse` subclass that is tied to a `SimpleCacheRe quest` and a `CacheControl`. In this example, shared references pass data from the request class to the response class. If you were storing responses in files, you'd just need to share the filenames instead. Along with the `SimpleCacheRequest` object from which it will read the data, you 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 and cache-control (if any) provided by the server for the cached representation of the resource.

Example 7-10. A concrete CacheResponse subclass

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

public class SimpleCacheResponse extends CacheResponse {

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

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

    this.request = request;
    this.control = control;
    this.expires = new Date(uc.getExpiration());
    this.headers = Collections.unmodifiableMap(uc.getHeaderFields());
  }

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

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

  public CacheControl getControl() {
    return control;
  }

  public boolean isExpired() {
    Date now = new Date();
    if (control.getMaxAge().before(now)) return true;
    else if (expires != null && control.getMaxAge() != null) {
      return expires.before(now);
    } else {
    return false;
    }
  }
}
```

Finally, you need a simple `ResponseCache` subclass that stores and retrieves the cached values as requested while paying attention to the original Cache-control header. Example 7-11 demonstrates such a simple class that stores a finite number of responses in memory in one big thread-safe `HashMap`. This class is suitable for a single-user, private cache (because it ignores the private and public attributes of Cache-control).

Example 7-11. An in-memory ResponseCache

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

public class MemoryCache extends ResponseCache {

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

  public MemoryCache() {
    this(100);
  }

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

  @Override
  public CacheRequest put(URI uri, URLConnection conn)
      throws IOException {

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

    CacheControl control = new CacheControl(conn.getHeaderField("Cache-Control"));
    if (control.noStore()) {
      return null;
    } else if (!conn.getHeaderField(0).startsWith("GET ")) { // only cache GET
      return null;
    }

    SimpleCacheRequest request = new SimpleCacheRequest();
    SimpleCacheResponse response = new SimpleCacheResponse(request, conn, control);

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

  @Override
  public CacheResponse get(URI uri, String requestMethod, Map<String, List<String>>
      requestHeaders)
      throws IOException {
```

```
      if ("GET".equals(requestMethod)) {
      SimpleCacheResponse response = responses.get(uri); // check expiration
        date
        if (response != null && response.isExpired()) {
        responses.remove(response);
          response = null;
        }
        return response;
      } else { return null;
      }
    }
}
```

Java only allows one URL cache at a time. To install or change the cache, 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 7-11 in an application:

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

Once a `ResponseCache` like Example 7-11 is installed, HTTP `URLConnection`s always use it.

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 docu- ments 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 docu- ment in the store was accessed and expunge only the oldest and least-used documents.

I've already mentioned that you could implement a cache on top of the filesystem instead of 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.

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 by- passes any local caching and downloads the file from the server afresh.

Because these fields are all protected, their values are accessed and modified via obvi- ously 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 (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.

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 de- faults will apply only to `URLConnection` objects constructed after the new default values are set.

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 7-12 opens a `URLConnection` to *http://www.oreilly.com/*, gets the `URL` of that connection, and prints it.

Example 7-12. Print the URL of a URLConnection to http://www.oreilly.com/

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

public class URLPrinter {

  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/
```

protected boolean connected

The boolean field `connected` is `true` if the connection is open and `false` if it's closed. Because 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 set- ting `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.

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`.

For example, this code fragment opens a connection that could ask the user for au- thentication 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 "Accessing Password-Protected Sites" on page 161.

Figure 7-1 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 clicking OK, then answering No when asked if you want to retry authorization—then `getInputStream()` will throw a `ProtocolEx ception`.

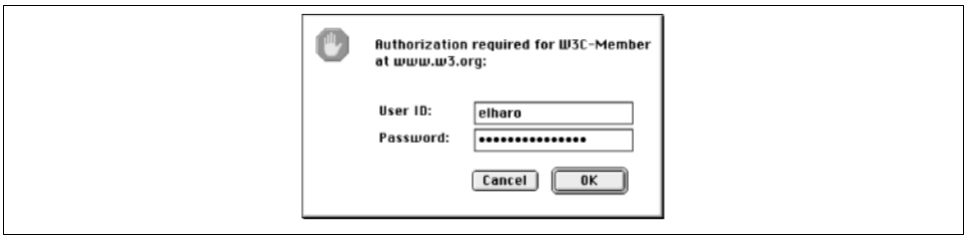

Figure 7-1. An authentication dialog

The static methods `getDefaultAllowUserInteraction()` and `setDefaultAllowUser Interaction()` determine the default behavior for `URLConnection` objects that have not set `allowUserInteraction` explicitly. Because 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 `setDefaul tAllowUserInteraction()` 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 allow- ing user interaction the default behavior:

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

protected boolean doInput

A `URLConnection` can be used for reading from a server, writing to a server, or both. The protected boolean field `doInput` is `true` if the `URLConnection` can be used for read- ing, `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:

```
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);
}
```

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 writing, `false` if it cannot be; it is `false` by default. To access this protected variable, use the `getDoOutput()` and `setDoOut put()` 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. We'll explore this in more detail later in "Writing Data to a Server" on page 218.

protected boolean ifModifiedSince

Many clients, especially web browsers and proxies, 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, 2014, Greenwich Mean Time:

```
GET / HTTP/1.1
Host: login.ibiblio.org:56452
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: close
If-Modified-Since: Fri, 31 Oct 2014 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:

```
HTTP/1.0 304 Not Modified
```

```
Server: WN/1.15.1
Date: Sun, 02 Nov 2014 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 millisec- onds 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 7-13 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 7-13. Set ifModifiedSince to 24 hours prior to now

```
import java.io.*;
import java.net.*;
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("Original if 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()));
            try (InputStream in = new BufferedInputStream(uc.getInputStream())) {
          Reader r = new InputStreamReader(in);
          int c;
          while ((c = r.read()) != -1) {
            System.out.print((char) c);
          }
          System.out.println();
        }
      } catch (IOException ex) {
        System.err.println(ex);
      }
    }
  }
}
```

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

```
% java Last24 http://www.elharo.com
Original if modified since: Wed Dec 31 19:00:00 EST 1969
Will retrieve file if it's modified since Sat Jun 01 11:11:27 EDT 2013
```

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

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. Stand- alone applications can use the

`java.net.ResponseCache` class. 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 docu- ment is retrieved by setting `useCaches` to `false`:

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

Timeouts

Four methods 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)
public int getConnectTimeout()
public void setReadTimeout(int timeout)
public int getReadTimeout()
```

The `setConnectTimeout()`/`getConnectTimeout()` methods control how long the sock- et 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 zero 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);
```

Configuring the Client Request HTTP Header

An HTTP client (e.g., a browser) sends the server a request line and a header. For example, here's an HTTP header that Chrome sends:

```
Accept:text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Charset:ISO-8859-1,utf-8;q=0.7,*;q=0.3
Accept-Encoding:gzip,deflate,sdch
Accept-Language:en-US,en;q=0.8
Cache-Control:max-age=0
Connection:keep-alive
Cookie:reddit_first=%7B%22firsttime%22%3A%20%22first%22%7D
DNT:1
Host:lesswrong.com
User-Agent:Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_3) AppleWebKit/537.31
```

```
(KHTML, like Gecko) Chrome/26.0.1410.65 Safari/537.31
```

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.

Each `URLConnection` sets a number of different name-value pairs in the header by de- fault. Here's the HTTP header that a connection from the `SourceViewer2` program of Example 7-1 sends:

```
User-Agent: Java/1.7.0_17
Host: httpbin.org
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: close
```

As you can see, it's a little simpler than the one Chrome 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.

You add headers to the HTTP header using the `setRequestProperty()` method before you open the connection:

```
public void setRequestProperty(String name, String value)
```

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` 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 a single named request property to have multiple values. In this case, the separate values will be separated by commas. For example, the Accept header sent by Java 7 in the previous code snippet has the four values text/html, image/gif, image/jpeg, and *.

For instance, web servers and clients store some limited persistent information with cookies. A cookie is a collection of name-value pairs. 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");
```

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)
```

There's no fixed list of legal headers. Servers usually ignore any headers they don't rec- ognize. HTTP does put a few restrictions on the content of the names and values of header fields. 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 `addRe questProperty()` 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 stan- dard getter method:

```
public String getRequestProperty(String name)
```

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

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

The keys are the header field names. The values are lists of property values. Both names and values are stored as strings.

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 `URLConnec tion` 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 dif- ferent 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
```

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.

Guessing MIME Media Types

If this were the best of all possible worlds, every protocol and every server would use standard MIME types to correctly specify the type of file being transferred. Unfortu- nately, 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)
```

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 con- ventions 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.

This method is not infallible by any means. For instance, it omits various XML appli- cations 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*). How- ever, it's a good start.

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

```
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 inspects the first 16 bytes of the `InputStream`, although sometimes fewer bytes are needed to make an identification. These guesses are often not as reliable as the guesses made by `guessContentTypeFromName()`. 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.

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. 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 match- ing 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.

Because 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 re- turned will be an instance of `HttpURLConnection`. Cast that `URLConnection` to `HttpURL Connection` like this:

```
URL u = new URL("http://lesswrong.com/");
URLConnection uc = u.openConnection();
HttpURLConnection http = (HttpURLConnection) uc;
```

Or, skipping a step, like this:

```
URL u = new URL("http://lesswrong.com/");
HttpURLConnection http = (HttpURLConnection) u.openConnection();
```

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 path of the resource that the client wants to retrieve and the version of the HTTP protocol that the client understands. For ex- ample:

```
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 docu- ment using `HEAD`:

```
HEAD /catalog/jfcnut/index.html HTTP/1.1
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`
- `DELETE`
- `OPTIONS`
- `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, `POST`ing 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.

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 7-15 is a simple program that uses the `HEAD` request method and prints the last time a file on a server was modified.

Example 7-15. Get the time when a URL was last changed

```
import java.io.*;
import java.net.*;
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())); }
     catch (MalformedURLException ex) {
       System.err.println(args[i] + " is not a URL I understand"); }
       catch (IOException ex) {
       System.err.println(ex);
     }
     System.out.println();
   }
  }
}
```

Here's the output from one run:

```
$ java LastModified http://www.ibiblio.org/xml/
http://www.ibiblio.org/xml/ was last modified at Tue Apr 06 07:45:29 EDT 2010
```

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

DELETE

The `DELETE` method removes a file at a specified URL from a web server. Because 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/2008march.html HTTP/1.1
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 authorization. For example:

```
HTTP/1.1 405 Method Not Allowed
Date: Sat, 04 May 2013 13:22:12 GMT
Server: Apache
Allow: GET,HEAD,POST,OPTIONS,TRACE
Content-Length: 334
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN"> <html><head>
<title>405 Method Not Allowed</title>
</head><body>
<h1>Method Not Allowed</h1>
<p>The requested method DELETE is not allowed for the URL
   /javafaq/2008march.html.</p>
<hr>
<address>Apache 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.

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 con- trasts 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 an editor might `PUT` a file on a web server:

```
PUT /blog/wp-app.php/service/pomdoros.html HTTP/1.1
```

```
Host: www.elharo.com
Authorization: Basic ZGFmZnk6c2VjZXJldA==
Content-Type: application/atom+xml;type=entry
Content-Length: 329
If-Match: "e180ee84f0671b1"

<?xml version="1.0" ?>
<entry xmlns="http://www.w3.org/2005/Atom"> <title>The Power of Pomodoros</title>
 <id>urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6a</id> <updated>2013-02-
 23T19:22:11Z</updated> <author><name>Elliotte Harold</name></author>
 <content>Until recently, I hadn't paid much attention to...</content> </entry>
```

As with deleting files, some sort of authentication is usually required and the server must be specially configured to support `PUT`. The details vary from server to server. Most web servers do not support `PUT` out of the box.

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
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 with:

```
HTTP/1.1 200 OK
Date: Sat, 04 May 2013 13:52:53 GMT
Server: Apache
Allow: GET,HEAD,POST,OPTIONS,TRACE
Content-Style-Type: text/css
Content-Length: 0
Connection: close
Content-Type: text/html; charset=utf-8
```

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 per- form on that URL.

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
Host: www.ibiblio.org
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: close
```

The server should respond like this:

```
HTTP/1.1 200 OK
Date: Sat, 04 May 2013 14:41:40 GMT
Server: Apache
Connection: close
Content-Type: message/http

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

The first five 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. However, if there were a proxy server between the client and server, it might not be.

Disconnecting from the Server

HTTP 1.1 supports persistent connections that allow multiple requests and responses to be sent over a single TCP socket. However, when Keep-Alive is used, the server won't immediately close a connection simply because it has sent the last byte of data to the client. The client may, after all, send another request. Servers will time out and close the connection in as little as 5 seconds of inactivity. However, it's still preferred for the client to close the connection as soon as it knows it's done.

The `HttpURLConnection` class transparently supports HTTP Keep-Alive unless you ex- plicitly turn it off. That is, it will reuse sockets if you connect to the same server again before the server has closed the connection. Once you know you're done talking to a particular host, the `disconnect()` method enables a client to break the connection:

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

If any streams are still open on this connection, `disconnect()` closes them. However, the reverse is not true. Closing a stream on a persistent connection does not close the socket and disconnect.

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
Cache-Control:max-age=3, must-revalidate
Connection:Keep-Alive
Content-Type:text/html; charset=UTF-8
Date:Sat, 04 May 2013 14:01:16 GMT
Keep-Alive:timeout=5, max=200
Server:Apache
Transfer-Encoding:chunked
Vary:Accept-Encoding,Cookie
WP-Super-Cache:Served supercache file from PHP

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

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

```
HTTP/1.1 404 Not Found
Date: Sat, 04 May 2013 14:05:43 GMT
Server: Apache
Last-Modified: Sat, 12 Jan 2013 00:19:15 GMT
ETag: "375933-2b9e-4d30c5cb0c6c0;4d02eaff53b80"
Accept-Ranges: bytes
Content-Length: 11166
Connection: close
Content-Type: text/html; charset=ISO-8859-1

<html>
<head>
<title>Lost ... and lost</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1"> </head>
<body bgcolor="#FFFFFF">
  <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
Connection: Keep-Alive
Content-Length: 299
Content-Type: text/html; charset=iso-8859-1
Date: Sat, 04 May 2013 14:20:58 GMT
Keep-Alive: timeout=5, max=200
Location: http://www.cafeaulait.org/
Server: Apache
```

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

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

The text string that follows the response code is called the *response message* and is returned by the aptly named `getResponseMessage()` method:

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

HTTP 1.0 defined 16 response codes. HTTP 1.1 expanded this to 40 different codes. Although 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 such as `HttpURLConnection.OK` and `HttpURLConnec tion.NOT_FOUND` representing the most common response codes. These are summar- ized in Table 6-1. Example 7-16 is a revised source viewer program that now includes the response message.

Example 7-16. A SourceViewer that includes the response code and message

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

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);
        }
        System.out.println();

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

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 specifically return that. In this ex- ample, you just fake it as "HTTP/1.x," like this:

```
% java SourceViewer3 http://www.oreilly.com HTTP/1.x 200 OK
Date: Sat, 04 May 2013 11:59:52 GMT
Server: Apache
Last-Modified: Sat, 04 May 2013 11:41:06 GMT
Accept-Ranges: bytes
Content-Length: 80165
Content-Type: text/html; charset=utf-8
Cache-Control: max-age=14400
Expires: Sat, 04 May 2013 15:59:52 GMT
Vary: Accept-Encoding
Keep-Alive: timeout=3, max=100
Connection: Keep-Alive
```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
...
```

However, `uc.getHeaderField(0)` does return the entire first HTTP request line, ver- sion included:

```
HTTP/1.1 200 OK
```

Error conditions

On occasion, the server encounters an error but returns useful information in the mes- sage body nonetheless. For example, when a client requests a nonexistent page from the *www.ibiblio.org* website, rather than simply returning a 404 error code, the server sends the search page shown in Figure 7-2 to help the user figure out where the missing page might have gone.

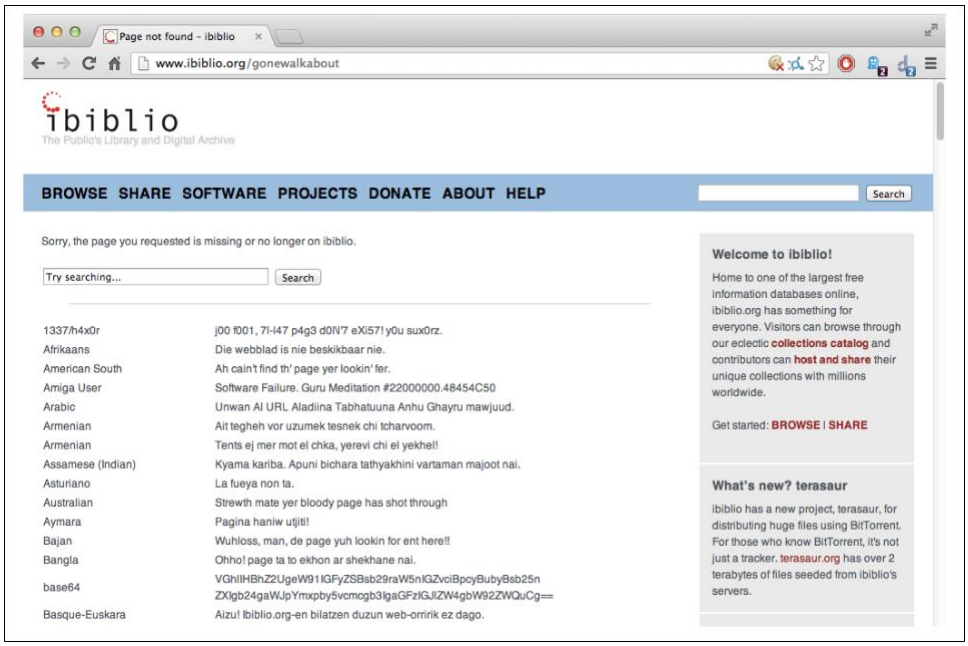

Figure 7-2. IBiblio's 404 page

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

```
public InputStream getErrorStream()
```

Generally, you'll invoke `getErrorStream()` inside a `catch` block after `getInput Stream()` has failed. Example 7-17 demonstrates with a program that reads form the input stream if possible. However, if that fails for any reason, it then reads from the error stream instead.

Example 7-17. Download a web page with a URLConnection

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

public class SourceViewer4 {
  public static void main (String[] args) {
    try {
      URL u = new URL(args[0]);
      HttpURLConnection uc = (HttpURLConnection) u.openConnection();
      try (InputStream raw = uc.getInputStream()) {
        printFromStream(raw);
      } catch (IOException ex) {
        printFromStream(uc.getErrorStream());
      }
    } catch (MalformedURLException ex) {
```

```
      System.err.println(args[0] + " is not a parseable URL");
    } catch (IOException ex) {
      System.err.println(ex);
    }
  }

  private static void printFromStream(InputStream raw) throws IOException {
  try (InputStream buffer = new BufferedInputStream(raw)) {
      Reader reader = new InputStreamReader(buffer);
      int c;
      while ((c = reader.read()) != -1) {
        System.out.print((char) c);
      }
    }
  }
}
```

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 `HttpURLConnec tion` class has two static methods that let you decide whether to follow redirects:

```
public static boolean getFollowRedirects()
public static voidsetFollowRedirects(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. Because these are static methods, they change the be- havior of all `HttpURLConnection` objects constructed after the method is invoked. The `setFollowRedirects()` method may throw a `SecurityException` if the security man- ager 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()
public void setInstanceFollowRedirects(boolean followRedirects)
```

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

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()
```

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.

Streaming Mode

Every request sent to an HTTP server has an HTTP header. One field in this header is the Content-length (i.e., 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 everything you write onto the `OutputStream` retrieved from the `HttpURLCon nection` 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 or 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 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, 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 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, 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 nonstreaming default is a bottleneck.

If you do happen to know the size of the request data in advance, you can 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)
public void setFixedLengthStreamingMode(long contentLength) // Java 7
```

Java will use this number in the 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 happens 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 `IllegalSta teException` 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 with 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.