Chapter 5

URLConnections

1. Opening URLConnections
2. Reading Data from a Server
3. Reading the Header
    i. Retrieving Specific Header Fields
    ii. Retrieving Arbitrary Header Fields
4. Caches
    i. Web Cache for Java
5. Configuring the Connection
    i. protected URL url
    ii. protected boolean connected
    iii. protected boolean allowUserInteraction
    iv. protected boolean doInput
    v. protected boolean doOutput
    vi. protected boolean ifModifiedSince
    vii. protected boolean useCaches
    viii. Timeouts
6. Configuring the Client Request HTTP Header
7. Writing Data to a Server
8. Security Considerations for URLConnections
9. Guessing MIME Media Types
10. HttpURLConnection

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:

1. It provides more control over the interaction with a server (especially an HTTP server) than the URL class.
   I. A URLConnection can inspect the header sent by the server and respond accordingly.
   II. It can set the header fields used in the client request.
   III. a URLConnection can send data back to a web server with HTTP methods(like GET, POST, etc).
2. the URLConnection class is part of Java's protocol handler mechanism, which also includes the URLStreamHandler class.

Opening URLConnections:

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

1. Construct a URL object i.e URL u = **new** URL("http://www.overcomingbias.com/");
2. Invoke the URL object's openConnection() method to retrieve a URLConnectionobject for that URL. URLConnection uc = u.openConnection();
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 metainformation, 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.
- When a URLConnection is first constructed, it is unconnected; that is, the local and remote host cannot send and receive data.
- 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.
- 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

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.

E.g. 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);
        }
}
```

Question: Differences between URL and URLConnection.

1. URLConnection provides access to the HTTP header.
2. URLConnection can configure the request parameters sent to the server **:** provides methods to set various request properties such as request method, request headers, cookies, and so on.
3. URLConnection can write data to the server as well as read data from the server **:** supports various HTTP methods such as POST, PUT, DELETE, and so on,

which can be used to send data to the server. It also provides methods to get the output stream.

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

The getHeaderField(String name) method returns the string value of a named header field.

Example:

```
Map<String, List<String>> hm = urlConn.getHeaderFields();
```

Point to be Noted:

• Names are case-insensitive.

• If the requested field is not present, null is returned.

• e.g. String lm = uc.getHeaderField("Last-modified");

Retrieving Specific Header

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

1. Content-type
2. Content-length
3. Content-encoding
4. Date
5. Last-modified
6. Expires

To get information about above 6 headers, there are Six Convenience Methods. These return the values of six particularly common header fields:

1. public int getContentLength()
2. public String getContentType()
3. public String getContentEncoding()
4. public long getExpiration()
5. public long getDate()
6. public long getLastModified()

Q. WAP to return above 6 headers.

```
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();
}
}
}
```

```
java HeaderViewer of 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
- 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.

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");
```

2. public String getHeaderFieldKey(int n)

- The keys of the header fields are returned by the getHeaderFieldKey(int n) method.
- The first field is 1, n^{th} field is n
- If a numbered key is not found, null is returned.
- You can use this in combination with getHeaderField() to loop through the complete header

- For example, in order to get the sixth KEY (not value) of the header of the URLConnec tion uc, you would write:

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

3. 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
- getHeaderFieldKey(int n) returns n^{th} key whereas getHeaderField(int n) returns value of n^{th} key.

Q. WAP to print all the headers present in a url

```
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:

```
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
```

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
- ➔ To convert the string to an integer, getHeaderFieldDate() uses the parseDate() method of java.util.Date

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.

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.
- 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

Cache:

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.

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 responses 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.

- 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 (entity tag 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"

i. 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

1. ResponseCache is an abstract class that defines the contract for a response cache. Its subclasses must provide the implementation for storing and retrieving cached responses.
2. CacheRequest is a concrete subclass of ResponseCache. It is responsible for writing a request to the cache and returning an OutputStream to write the request body to. It also provides a method to abort the request if necessary.
3. CacheResponse is a concrete subclass of ResponseCache. It represents a cached response and

provides methods to retrieve the response headers and body as an InputStream.

Two abstract methods in the ResponseCache class store and retrieve data from the system'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

Example 7-7. The CacheRequest class

```
package java.net;

public abstract class CacheRequest {
public abstract OutputStream getBody() throws IOException;
public abstract void abort();
}
```

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();
```

}

}

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;
}
```

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

Eg:ResponseCache.setDefault(new MemoryCache());

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;
```

- **url**: This variable represents the URL for which the connection is being established. It is declared as **protected** to allow subclasses to access it directly.
- **doInput**: This variable is a boolean flag indicating whether the connection allows input. By default, it is set to **true**, meaning that input is allowed.

- **doOutput**: This variable is a boolean flag indicating whether the connection allows output. By default, it is set to **false**, meaning that output is not allowed.
- **allowUserInteraction**: This variable is a boolean flag indicating whether the connection allows user interaction. By default, it is set to the value of the **defaultAllowUserInteraction** variable, which is defined in the **URLConnection** class.
- **useCaches**: This variable is a boolean flag indicating whether the connection uses caches. By default, it is set to the value of the **defaultUseCaches** variable, which is defined in the **URLConnection** class.
- **ifModifiedSince**: This variable is a long value representing the timestamp of the last modified date for the resource being accessed. By default, it is set to 0, meaning that no modification date is specified.
- **connected**: This variable is a boolean flag indicating whether the connection is currently open. By default, it is set to **false**, meaning that the connection is closed.

Because 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()

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

public boolean getDefaultUseCaches() True

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)

Example. 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);
}
}
}
```

O/P:

http://www.oreilly.com/

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()

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");
URLConnection uc = u.openConnection();
uc.setConnectTimeout(30000);
uc.setReadTimeout(45000);
```

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;

Accept-Charset:ISO-8859-1,utf-8;

Accept-Encoding:gzip,deflate,sdch

Accept-Language:en-US,en;

Cache-Control:max-age=0

Connection:keep-alive

Cookie:reddit_first=%7B%22firsttime%22%3A%20%22first%22%7D

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

✓public void setRequestProperty(String name, String value)

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

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;

✓ public void addRequestProperty(String name, String value)

✓ public String getRequestProperty(String name)

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

- **setRequestProperty(String name, String value)** method sets the value of a request header field with the specified name. If a header with the same name already exists, its value is replaced with the new value.
- **addRequestProperty(String name, String value)** method adds a new HTTP header with the specified name and value to the request. If a header with the same name already exists, the new value is appended to the existing value using a comma as a separator.
- **getRequestProperty(String name)** method is used to retrieve the value of the request header with the given name from the URL connection. If the header is not set, this method returns null.
- **getRequestProperties()** method returns a map containing all the request headers set on the URL connection.

Writing Data to a Server:

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.

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.

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.

Guessing MIME Media Types:

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.

For example, if you pass the string "example.html" to the guessContentTypeFromName method, it will return the string "text/html", which is the MIME type typically associated with HTML files. If you pass the string "example.png", it will return the string "image/png", which

is the MIME type typically associated with PNG image files.

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).

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 example, an XML document that begins with a comment rather than an XML declaration would be mislabeled as an HTML file.

HttpURLConnection:

1. The Request Method
2. Disconnecting from the Server
3. Handling Server Responses
4. Proxies
5. Streaming Mode

HttpURLConnection:

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

Cast that URLConnection to HttpURLConnection 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();
```

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 example:

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

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

Host: www.oreilly.com

Accept: text/html, image/gif, image/jpeg,

Connection: close

By default, HttpURLConnection uses the GET method.

We 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

Head:

```
//Print the last modified date of a given URL
import java.io.*;
import java.net.*;
import java.util.*;

public class HeaderLastModify {
	public static void main(String[] args) {
		String url= "http://www.ibiblio.org/xml/";
			try {
				URL u = new URL(url);
				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(url + " is not a URL I understand");
			} catch (IOException ex) {
				System.err.println(ex);
			}
```

```
                System.out.println();
            }

}
```

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

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: 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

Methods for Handling Server Response

public int getResponseCode() throws IOException

public String getResponseMessage() throws IOException

// Print the response code and message from a given URL

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

public class ResponseCodeMsg {
    public static void main(String[] args) {

        try {
            // Open the URLConnection for reading
            URL u = new URL("https://example.com/dfdfdf/");
            HttpURLConnection uc = (HttpURLConnection) u.openConnection();
            int code = uc.getResponseCode();
```

```
                String response = uc.getResponseMessage();
                System.out.println("Code: " + code + "\n" + "Response
Message: " + response);
            } catch (MalformedURLException ex) {
                System.err.println(args[0] + " is not a parseable URL");
            } catch (IOException ex) {
                System.err.println(ex);
            }
        }
}
```

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

➢ 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.

➢ But, 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)

➢ 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 and later