NETWORK PROGRAMMING

CHAPTER 5 : URLCONNECTIONS

CHANDAN GUPTA BHAGAT

https://me.chandanbhagat.com.np

CONTENT

- Opening URLConnections
- Reading Data from a Server
- Reading the Header : Retrieving Specific Header Fields, Retrieving Arbitrary Header Fields
- Caches : Web Cache for Java
- Configuring the Connection : protected URL url, protected boolean connected, protected boolean allowUserInteraction, protected boolean doInput, protected boolean doOutput, protected boolean ifModifiedSince, protected boolean useCaches, Timeouts
- Configuring the Client Request HTTP Header
- Writing Data to a Server
- Security Considerations for URLConnections
- Guessing MIME Media Types
- HttpURLConnection, The Request Method, Disconnecting from the Server, Handling Server Responses, Proxies, Streaming Mode

URLConnections

- It is an abstract class that represents an active connection to a resource specified by a URL.
- 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, It 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 two subclasses HttpURLConnection and JarURLConnection makes the connetion between the client Java program and URL resource on the internet.
- With the help of URLConnection class, a user can read and write to and from any resource referenced by an URL object.
- Once a connection is established and the Java program has an URLConnection object, we can use it to read or write or get further information like content length, etc.

Opening URLConnections

Opening URLConnections

- A program that uses the URLConnection class directly follows this basic sequence of steps:
  a. Construct a URL object.
  b. Invoke the URL object's openConnection() method to retrieve a URLConnection object for that URL.
  c. Configure the URLConnection.
  d. Read the header fields.
  e. Get an input stream and read data.
  f. Get an output stream and write data.
  g. Close the connection.

Opening URLConnections

- 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 open Connection() method of the URL class.

```
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 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).

Opening URLConnections

- Example, a sun.net.www.protocol.file.FileURLConnection's connect() method converts the URL to a filename in the appropriate directory, creates MIME information for the file, and then opens a buffered FileInputStream to the file.
- The connect() method of sun.net.www.protocol.http.HttpURLConnection creates a sun.net.www.http.HttpClient object, which is responsible for connecting to the server:

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

- When a URLConnection is first constructed, it is unconnected; that is, the local and remote host cannot send and receive data. There is no socket connecting the two hosts.
- The connect() method establishes a connection—normally using TCP sockets but possibly through some other mechanism between the local and remote host so they can send and receive data.
- However, getInputStream(), getContent(), 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 Server

Reading Data From Server

- The following is the minimal set of steps needed to retrieve data from a URL using a URLConnection object:
  a. Construct a URL object.
  b. Invoke the URL object's openConnection() method to retrieve a URLConnection object for that URL.
  c. Invoke the URLConnection's getInputStream() method.
  d. 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.
- The openStream() method of the URL class just returns an InputStream from its own URLConnection object.
- 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.

```java
package Test;

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

public class Viewer {
    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);
            }
        }
    }
}
```

Reading The Header

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.

Reading the Header

- 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

Retrieving Specific Header Fields

- **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. 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
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 previous code by using the encoding specified in the HTTP header to decode the document, or ISO-8859-1 (the HTTP default) if no such encoding is specified. If a nontext type is encountered, an exception is thrown.

Retrieving Specific Header Fields

```
package Test;

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);
            }
        }
    }
}
```

Retrieving Specific Header Fields

- **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, getContentLength() 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
```

- HTTP servers don't always close the connection exactly where the data is finished; therefore, you don't know when to stop reading. To download a binary file, it is more reliable to use a URLConnection's getContentLength() method to find the file's length, then read exactly the number of bytes indicated.

Retrieving Specific Header Fields

- **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, getContentLength() 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
```

- HTTP servers don't always close the connection exactly where the data is finished; therefore, you don't know when to stop reading. To download a binary file, it is more reliable to use a URLConnection's getContentLength() method to find the file's length, then read exactly the number of bytes indicated.

Retrieving Specific Header Fields

```
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);
        }
    }
}
```

Retrieving Specific Header Fields

```
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.");
    }
    extractedStream(u, uc, contentLength);
}
```

Retrieving Specific Header Fields

```
private static void extractedStream(URL u, URLConnection uc, int
 contentLength)
        throws IOException, FileNotFoundException {
    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();
        }
    }
}
```

Retrieving Specific Header Fields

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

Retrieving Specific Header Fields

- **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.

Retrieving Specific Header Fields

- **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.

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 URLConnec tion object uc, you could write:

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

Retrieving Arbitrary Header Fields

- 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(), getEx pirationDate(), 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., Contentlength 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);
```

Retrieving Arbitrary Header Fields

- **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.
- Besides the headers with named getter methods, this server also provides Server, AcceptRanges, 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.

Retrieving Arbitrary Header Fields

- 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

Retrieving Arbitrary Header Fields

- **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

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.
- HTTP headers can adjust
  - Expires Header : Ok to cache this till time
  - Cache-Control Header :
    - 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.
  - Last Modified Header : date when the resource was last changed.
  - Etag Header : unique identifier for the resource that changes when the resource does.

Web Caches 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 CacheRequest and CacheResponse by passing it to the static method ResponseCache.setDefault(). 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 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

- 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

Web Caches for Java : 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.

Web Caches for Java : Concrete CacheRequest Class

```
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();
    }
}
```

Web Caches for Java : CacheResponse

- 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. 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.

```
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.
- `ResponseCache.setDefault(new MemoryCache());`
- Once a ResponseCache like is installed, HTTP URLConnections always use it.

Web Caches for Java : CacheResponse

- 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. 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.

```
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.
- `ResponseCache.setDefault(new MemoryCache());`
- Once a ResponseCache like is installed, HTTP URLConnections always use it.

Configuring the Connection

Configuring the Connection

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

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

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

Configuring the Connection

```
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.

Configuring the Connection

- There are also some getter and setter methods that define the default behavior for all instances of URLConnection. These are
- **public boolean** getDefaultUseCaches()
  public boolean setDefaultUseCaches(**boolean** defaultUseCaches)
  public boolean setDefaultAllowUserInteraction(**boolean** defaultAllowUserInteraction)
  public static boolean getDefaultAllowUserInteraction()
  public static FileNameMap getFileNameMap()
  public static void setFileNameMap(**FileNameMap** map)

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.

```
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);
        }
    }
}
```

-

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

protected boolean allowUserInteraction

- Some URLConnections 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.

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

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 reading, 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()
```

- 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);
}
```

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
```

protected boolean ifModifiedSince

- The client then loads the document from its cache. Not all web servers respect the IfModified-Since field. Some will send the document whether it’s changed or not. The ifModifiedSince field in the URLConnection class specifies the date (in milliseconds since midnight, Greenwich Mean Time, January 1, 1970), which will be placed in the If-Modified-Since header field. Because ifModifiedSince is protected, programs should call the getIfModifiedSince() and setIfModifiedSince() methods to read or modify it:
- **public long** getIfModifiedSince()
  public void setIfModifiedSince(**long** ifModifiedSince)

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. Standalone 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 document 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);
}
```

protected boolean useCaches

- 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, URLConnections 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 socket waits for the initial connection. The setReadTimeout()/getReadTimeout() methods control how long the input stream waits for data to arrive. All four methods measure timeouts in milliseconds. All four interpret 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

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.

Configuring the Client Request HTTP Header

- It's important to understand that this is *not the HTTP header that the server sends to the client* that is read by the various getHeader Field() and getHeaderFieldKey() methods discussed previously. This is the *HTTP header that the client sends to the server*.
- Each URLConnection sets a number of different name-value pairs in the header by default. Here's the HTTP header that a connection from the SourceViewer2 program of 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.

Configuring the Client Request HTTP Header

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

Configuring the Client Request HTTP Header

- 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 recognize. 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 addRequestProperty() throw an IllegalArgumentException. Otherwise, it's quite easy to make a URLConnection send malformed headers to the server, so be careful. Some servers will handle the malformed headers gracefully. Some will ignore the bad header and return the requested document anyway, but some will reply with an HTTP 400, Bad Request error. If, for some reason, you need to inspect the headers in a URLConnection, there's a standard getter method:

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

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

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

Writing Data to a Server

Writing Data to a Server

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

```
public OutputStream getOutputStream()
```

- A URLConnection doesn't allow output by default, so you have to call setDoOut put(true) before asking for an output stream. When you set doOutput to true for an *http* URL, the request method is changed from GET to POST.
- However, GET should be limited to safe operations, such as search requests or page navigation, and not used for unsafe operations that create or modify a resource, such as posting a comment on a web page or ordering a pizza. Safe operations can be bookmarked, cached, spidered, prefetched, and so on. Unsafe operations should not be. Once you have an OutputStream, buffer it by chaining it to a BufferedOutputStream or a BufferedWriter. You may also chain it to a DataOutputStream, an OutputStream Writer, or some other class that's more convenient to use than a raw OutputStream.

Writing Data to a Server

```
try {
        URL u = new URL("http://www.somehost.com/cgi-bin/acgi");
        // open the connection and prepare it to POST
        URLConnection uc = u.openConnection();
        uc.setDoOutput(true);
        OutputStream raw = uc.getOutputStream();
        OutputStream buffered = new BufferedOutputStream(raw);
        OutputStreamWriter out = new OutputStreamWriter(buffered, "8859_1");
        out.write("first=Julie&middle=&last=Harting&work=String+Quartet\r\n");
        out.flush();
        out.close();
} catch (IOException ex) {
        System.err.println(ex);
}
```

Writing Data to a Server

- Sending data with POST is almost as easy as with GET. Invoke setDoOutput(true) and use the URLConnection's getOutputStream() method to write the query string rather than attaching it to the URL. Java buffers all the data written onto the output stream until the stream is closed. This enables it to calculate the value for the Content-length header.
- The query string is built using the add() method. The post() method actually sends the data to the server by opening a URLConnection to the specified URL, setting its doOutput field to true, and writing the query string on the output stream. It then returns the input stream containing the server's response.
- The post() method is the heart of the class. It first opens a connection to the URL stored in the url field. It sets the doOutput field of this connection to true because this URL Connection needs to send output and chains the OutputStream for this URL to an ASCII OutputStreamWriter that sends the data, then flushes and closes the stream. *Do not forget to close the stream!* If the stream isn't closed, no data will be sent. Finally, the URLConnection's InputStream is returned.

Writing Data to a Server

- To summarize, posting data to a form requires these steps:
  1. Decide what name-value pairs you'll send to the server-side program.
  2. Write the server-side program that will accept and process the request. If it doesn't use any custom data encoding, you can test this program using a regular HTML form and a web browser.
  3. Create a query string in your Java program. The string should look like this:
     name1=value1&name2=value2&name3=value3
     Pass each name and value in the query string to URLEncoder.encode() before adding it to the query string.
  4. Open a URLConnection to the URL of the program that will accept the data.
  5. Set doOutput to true by invoking setDoOutput(true).
  6. Write the query string onto the URLConnection's OutputStream.
  7. Close the URLConnection's OutputStream.
  8. Read the server response from the URLConnection's InputStream.
- GET should only be used for safe operations that can be bookmarked and linked to. POST should be used for unsafe operations that should not be bookmarked or linked to. The getOutputStream() method is also used for the PUT request method, a means of storing files on a web server. The data to be stored is written onto the OutputStream that getOutputStream() returns. However, this can be done only from within the HttpURLConnection subclass of URLConnection, so discussion of PUT will have to wait a little while.

Security Considerations for URLConnections

Security Considerations for URLConnections

- URLConnection objects are subject to all the usual security restrictions about making network connections, reading or writing files, and so forth. For instance, a URLConnection can be created by an untrusted applet only if the URLConnection is pointing to the host that the applet came from. However, the details can be a little tricky because different URL schemes and their corresponding connections can have different security implications. For example, a *jar* URL that points into the applet's own *jar* file should be fine. However, a file URL that points to a local hard drive should not be.
- Before attempting to connect a URL, you may want to know whether the connection will be allowed. For this purpose, the URLConnection class has a getPermission() method:

```
public Permission getPermission() throws IOException
```

- 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

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. Unfortunately, that's not the case. Not only do we have to deal with older protocols such as FTP that predate MIME, but many HTTP servers that should use MIME don't provide MIME headers at all or lie and provide headers that are incorrect (usually because the server has been misconfigured). The URLConnection class provides two static methods to help programs figure out the MIME type of some data; you can use these if the content type just isn't available or if you have reason to believe that the content type you're given isn't correct. The first of these is URLConnection.guessContentTypeFromName():
- **public static** String guessContentTypeFromName(String name)
- This method tries to guess the content type of an object based upon the extension in the filename portion of the object's URL. It returns its best guess about the content type as a String. This guess is likely to be correct; people follow some fairly regular conventions when thinking up filenames. The guesses are determined by the *content-types.properties* file, normally located in the *jre/lib* directory. On Unix, Java may also look at the *mailcap* file to help it guess.

Guessing MIME media types

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

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

- This method tries to guess the content type by looking at the first few bytes of data in the stream. For this method to work, the InputStream must support marking so that you can return to the beginning of the stream after the first bytes have been read. Java 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

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 matching the various HTTP response codes. Finally, it overrides the getPermission() method from the URLConnection superclass, although it doesn't change the semantics of this method at all. 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 returned 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 example:

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

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

```
HEAD /catalog/jfcnut/index.html HTTP/1.1
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 Request Method

- If it’s some other method, then a java.net.ProtocolException, a subclass of IOException, is thrown. However, it’s generally not enough to simply set the request method. Depending on what you’re trying to do, you may need to adjust the HTTP header and provide a message body as well. For instance, POSTing a form requires you to provide a Content-length header. We’ve already explored the GET and POST methods.
- Let’s look at the other five possibilities. Some web servers support additional, nonstandard request methods. For instance, WebDAV requires servers to support PROPFIND, PROP PATCH, MKCOL, COPY, MOVE, LOCK, and UNLOCK. However, Java doesn’t support any of these.

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, indicating that the URL you requested no longer points to a document.

Handling Server Responses

- 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
```

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 HttpURLConnection until the stream is closed. At that point, it knows how many bytes are in the body so it has enough information to write the Content-length header.
- This scheme is fine for small requests sent in response to typical web forms. However, it's burdensome for responses to very long forms or some SOAP messages. It's very wasteful and slow for medium 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)
```

Streaming Mode

- 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
```

- Because this number can actually be larger than the maximum size of an int, in Java 7 and later you can use a long instead.

Streaming Mode

- 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 IllegalStateException if the connection is connected or has already been set to chunked transfer encoding. (You can't use both chunked transfer encoding and fixed-length streaming mode on the same request.)"
- Fixed-length streaming mode is transparent on the server side. Servers neither know nor care how the Content-length was set, as long as it's correct. However, like chunked transfer encoding, streaming mode does interfere 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.

THANK YOU

YOU CAN FIND THIS SLIDES/NOTES IN

https://chandanbhagat.com.np/docs/network-programming/

AND CODES IN

https://github.com/chandan-g-bhagat/network-programming