Unit-5

URL Connection:

Comparision:

- **URL**: Mainly deals with parsing and handling the URL string.
- **URLConnection**: Deals with the communication with the URL resource, allowing you to **send requests** and **receive responses**.

The **URLConnection** class contains many **methods** that let you communicate with the URL over the network.

URLConnection is an **HTTP-centric** class, many of its methods are useful when working with HTTP URLs.

This **URLConnection** class helps read and write the data to the specific/specified resource, which is actually referred to by an URL.

URLConnection can send data back to a web server with POST, PUT, and other HTTP request methods

Opening URLConnection:

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

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

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

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

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

```
        }
```

```
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.

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

Lab: Write a java program to fetch Website content using URLConnection Class

Example:

```
import java.io.*;

import java.net.*;
public class App
    {
    public static void main(String[] args) throws Exception {
        StringBuilder content = new StringBuilder();
        try
        {
          URL u = new URL("https://samriddhicollege.edu.np/contact-
us/");
          URLConnection uc = u.openConnection();
          InputStream in = uc.getInputStream();
         BufferedReader bufferedReader = new BufferedReader(new
      InputStreamReader(in));
          String line;
          while ((line = bufferedReader.readLine()) != null)
          {
            content.append(line + "\n");
          }
          bufferedReader.close();
        }
        catch(Exception e)
        {
          e.printStackTrace();
        }
        System.out.println(content);

    }
}
```

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

Reading the Header:

1. The **getHeaderField**(String name) method returns the string value of a named header field.
2. Names are case-insensitive.
3. If the requested field is not present, null is returned.
4. String lm = uc.getHeaderField("Last-modified");

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
```

Lab: Write a java program to fetch all HTTP header Fields using URLConnection

Get Header Fields

```
URL u = new URL("https://samriddhicollege.edu.np/contact-us/");
          URLConnection uc = u.openConnection();
          Map<String, List<String>> header = uc.getHeaderFields();
            for (Map.Entry<String, List<String>> mp1 : header.entrySet())
            {
            System.out.print(mp1.getKey() + " : ");
            System.out.println(mp1.getValue().toString());
            }
```

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

Six Convenience Methods

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

Write a Java program to retrieving Specific Header Fields from URL using URLConnection Class.

```
import java.net.*;
import java.sql.*;
public class App {
    public static void main(String[] args) throws Exception {

          URL u = new URL("https://samriddhicollege.edu.np/contact-us/");
          URLConnection uc = u.openConnection();
            System.out.println("Content-type: " +
uc.getContentType());
            System.out.println("Content-encoding: " + uc.getContentEncoding());
            System.out.println("Date: " + new Date(uc.getDate()));
            System.out.println("Last modified: " + new
Date(uc.getLastModified()));
            System.out.println("Expiration date: "+ new
Date(uc.getExpiration()));
            System.out.println("Content-length: " + uc.getContentLength());

    }
}
```

Retrieving Arbitrary Header Fields:

- **Custom headers** (**Arbitrary Header**) that can be included in HTTP requests and responses in addition to the standard headers defined by the HTTP/1.1 specification.
- These custom headers are often used to pass additional information between the client and the server that is not covered by the standard headers.
- 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.

1. `public String getHeaderField(String name)`

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

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

2. `public String getHeaderFieldKey(int n)`

Example:

```
import java.sql.*;
public class App {
    public static void main(String[] args) throws Exception {

          URL u = new URL("https://samriddhicollege.edu.np/contact-us/");
          URLConnection uc = u.openConnection();
          String contentType = uc.getHeaderField("Server");
          System.out.println(contentType);
          String headerkey = uc.getHeaderFieldKey(1);
          System.out.println(headerkey);
    }
}
```

Web Cache:

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 not cached. However, HTTP headers can adjust this:

- The Cache-control header (HTTP 1.1) offers fine-grained cache policies**:(some common directives used with the Cache-Control header**)
  - **Public:** *Cache-Control: public* - The response may be cached by any cache.
  - **Private:** *Cache-Control: private* - The response is intended for a single user and must not be cached by shared caches.
  - **No-Cache:** *Cache-Control: no-cache* - The cache must validate the response with the origin server before using it.
  - **No-Store:** *Cache-Control: no-store* - The cache must not store any part of the request or response.
  - **Max-Age:** *Cache-Control: max-age=3600* - The response is considered fresh for 3600 seconds.
  - **S-Maxage:** *Cache-Control: s-maxage=3600* - The shared cache is considered fresh for 3600 seconds, overriding max-age.

- The **Last-modified** header is the date when the resource was last changed.
- The **ETag** header (HTTP 1.1) is a unique identifier for the resource that changes when the resource does.

```
Example:
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"
```

- Web caching is the activity of storing data for reuse, such as a computer of a web page served by a web server
- It is cached or stored the first time a user visits the page and the next time a user requests the same page, a cache will serve the copy, which helps keep the origin server from getting overloaded.
- By default, java does not cache anything. To install a system-wide cache of the **URL** class will use, you need the following:
    - **A concreate subclass of ResponseCache**
    - **A concrete subclass of CacheRequest**
    - **A concrete subclass of CacheResponse**

Q: What is web cache? List some common directives used with the Cache-Control header

Configuring the Connection:

Protected URL url:

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**.

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()
```

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. **getURL()** method return URLConnection's URL field .

```
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.
false if it's closed.Any method that causes the **URLConnection** to connect should set this variable to **true**

protected boolean allowUserInteraction

As its name suggests, the **allowUserInteraction** field specifies whether user interaction is allowed. It is **false** by default.

```
try {
        URL u = new URL("http://www.example.com/passwordProtectedPage.html");
        URLConnection uc = u.openConnection();
        uc.setAllowUserInteraction(true);
        InputStream in = uc.getInputStream();
        } catch (IOException ex) {
        System.err.println(ex);
        }
```

protected boolean doInput

- A **URLConnection** can be used for **reading** from a server, **writing** to a server, or both. The default is true.
- The protected **boolean** field **doInput** is true if the **URLConnection** can be used for reading, false if it cannot be.
- To access this protected variable, use the public **getDoInput**() and **setDoInput**() methods:

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

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.

```
  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 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.

protected boolean useCaches

Some clients, notably web browsers, can retrieve a document from a **local cache**, rather than retrieving it from a **server**.

- The default value is **true**, meaning that the **cache** will be used;
- **false** means the cache won't be used.

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

Example:
 try {
        URL u = new URL("http://www.sourcebot.com/");
        URLConnection uc = u.openConnection();
        uc.setUseCaches(false);//cache won't be used
        // read the document...
        } catch (IOException ex) {
        System.err.println(ex);
        }
```

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()
```

Example:

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

Configuring the Client Request HTTP Header

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

```
Accept:text/html,application/xhtml+xml,application/xml;q=0.9,*/*
;q=0.8
Accept-Charset:ISO-8859-1,utf-8;q=0.7,*;q=0.3
Accept-Encoding:gzip,deflate,sdch
Accept-Language:en-US,en;q=0.8
Cache-Control:max-age=0
Connection:keep-alive
Cookie:reddit_first=%7B%22firsttime%22%3A%20%22first%22%7D
DNT:1
Host:lesswrong.com
User-Agent:Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_3)
AppleWebKit/537.31
(KHTML, like Gecko) Chrome/26.0.1410.65 Safari/537.31
```

A web server can use this information to serve different pages to different clients, to get and set cookies, to authenticate users through passwords, and more.

Each `URLConnection` sets a number of different name-value pairs in the header by default.

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. Example:

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

The headers in a **URLConnection**, there's a standard getter method:

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

Security Considerations for URLConnections

- `URLConnection` objects are subject to all the usual security restrictions about making network connections, reading or writing files, and so on.
- 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.
- 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

- MIME types let the browser know what each file is. Browsers associate other applications, helper applications/extensions to handle certain MIME types. So, setting the correct MIME types will let the browser handle the object/file the way it was meant to be done.
- Every protocol and every server would use the MIME typing method to specify what kind of file it was transferring.

- The `URLConnection` class provides two static methods to help programs figure out the MIME type.
    - The first of these `URLConnection.guessContentTypeFromName();`
        - `protected static String guessContentTypeFromName(String name)`
        - This above 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 `String`.
        -
    - 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.

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.

Because this class is abstract and its only constructor is protected, you can't directly create instances of `HttpURLConnection`.

Cast that `URLConnection` to `HttpURLConnection` like this

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

The Request Method

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

GET

• POST

• HEAD

• PUT

• DELETE

• OPTIONS

• TRACE

```
import java.net.*;
import java.sql.Date;
import java.io.*;
public class App
    {
    public static void main(String[] args) throws Exception {
        try {
        URL u = new URL("https://buddhimalla.com/");
        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("https://buddhimalla.com/" + " is not a URL I
understand");
      } catch (IOException ex) {
      System.err.println(ex);
      }
      System.out.println();

    }
}
```

DELETE

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

```
DELETE /javafaq/2008march.html HTTP/1.1
Host: www.ibiblio.org
```

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

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 HttpURLConnection class keep supports HTTP Keep-Alive unless you explicitly turn it off.

the **disconnect()** method enables a client to break the connection

```
public abstract void 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
```

404 Status code Not Found 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
```

Moved Permanently 301 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
```

Proxies

Many users behind firewalls or using AOL(America Online) 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.

- What is the purpose of the CookieStore class in Java?

- Explain the role of the HttpCookie class.
- Describe the significance of the Max-Age and Secure attributes of a cookie.
- How would you handle cookies that need to be associated with multiple URIs?

  Provide an example based on the provided code.