Unit 5:
URLConnections

By

Khushbu Kumari Sarraf

EMBA,BE

URL

- A URL (Uniform Resource Locator) is a unique identifier used to locate a resource on the Internet. It is also referred to as a web address.

Examples:

http://www.c-sharpcorner.com/authors/fd0172/sandeep-sharma.aspx

Most URL can be broken into 4 parts:

- Protocol-In this case, HTTP is the protocol.
- Host-In this case, www.c-sharpcorner.com is the server name.
- Port-It is an optional attribute, If we write http://www.c-sharpcorner.com:90/authors/fd0172/, 90 is the port number.
- File path-In this case, /authors/fd0172/sandeep-sharma.aspx is the file name.

URL Class

- Java URL class is the gateway to access the resources on the web. The object of the **java.net.URL** class represents the URL and this object manages all the information present in the URL string. There are many methods in the Java URL class to create the object of the URL class **Class Constructors:**

S.N	Constructor	Description
1	URL(String address) throwsMalformedURLException	It creates a URL object from the given input String.
2	URL(String protocol, String host, String file)	This constructor creates a URL object from the given input protocol, host, and file name.
3	URL(String protocol, String host, int port, String file)	This constructor creates a URL object from the specified protocol, hostname, port number and filename.
4	URL(URL context, String spec)	It makes a URL object by parsing the given String spec in the given context.
5	URL(String protocol, String host, int port, String file, URLStreamHandler handler)	This constructor makes a URL object from the specified protocol, hostname, port number, file, and handler.
6	URL(URL context, String spec, URLStreamHandler handler)	Makes a URL by parsing the given spec with the given input handler inside the given context.

URL Class

Some commonly used public methods of the URL class are:

S.N	Method Name	Description
1	public String toString()	The toString() method returns the given URL object in the string form.
2	public String getPath()	This method returns the path of the URL. It returns null if the URL is empty.
3	public String getQuery()	This method gives the query part of the URL. A query is a part of the URL after the '?' in the URL.
4	public String getAuthority()	This method returns the authority part of the URL and null if it is empty.
5	public String getHost()	This method gives the hostname associated with the URL in IPv6 format
6	public String getFile()	This method returns the filename of the URL.
7	Public int getPort()	This method returns the port number of the URL.
8	Public int getDefaultPort()	This method returns the default port number used by the URL.
9	Public String getRef()	This method returns the reference to the URL object. The reference is the part represented by '#' in the URL.
10	Public String getProtocol()	It returns the protocol associated with the URL.

URL Class

Code to Create a URL using URL Components:

```
import java.net.MalformedURLException;
import java.net.URL;
public class URLClassDemo {
public static void main(String[ ] args) throws MalformedURLException {
String protocol = "http";
String host = “example.com";
String file = "/tutorials/java-method-overriding/";
 URL url = new URL(protocol, host, file); System.out.println("URL is: " +url.toString()); }
}
```

Output:

URL is:
http://example.com/tutorials/java-method-overriding/

URL Class

Code to Create a URL using URL Components:

```
import java.net.MalformedURLException;
import java.net.URL;
public class URLClassDemo {
 public static void main(String[] args) throws MalformedURLException
{ URL url1 = new URL("https://example.com/tutorials/java-
polymorphism/");
System.out.println("url1 is: " +url1.toString());
 System.out.println("\nDifferent components of the url1");
System.out.println("Protocol: " + url1.getProtocol());
System.out.println("Hostname: " + url1.getHost());
System.out.println("Port: " + url1.getPort());
 System.out.println("Default port: " + url1.getDefaultPort());
 System.out.println("Query: " + url1.getQuery());
 System.out.println("Path: " + url1.getPath());
System.out.println("File: " + url1.getFile());
System.out.println("Reference: " + url1.getRef());
System.out.println("Authority: " + url1.getAuthority());
```

Output

```
url1 is:
https://example.com/tutorials/java
-polymorphism/Different
components of the url1
Protocol: https
Hostname: techvidvan.com
Port: -1
Default port: 443
Query: null
Path: /tutorials/java-
polymorphism/
File: /tutorials/java-polymorphism/
Reference: null
Authority: example.com
```

URLConnections

- **URLConnection Class** in Java is an abstract class that represents a connection of a resource as specified by the corresponding URL. It is imported by the *java.net* package. The URLConnection class is utilized for serving two different related purposes:
- Firstly it provides control on interaction with a server(especially an HTTP server) than URL class.
- Secondly, with a URLConnection we can check the header sent by the server and respond accordingly, we can configure header fields used in client requests. We can also download binary files by using URLConnection.

There are mainly two subclasses that extends the URLConnection classes:

1. HttpURLConnection-If we are connecting to any url which uses “http” as its protocol, then HttpURLConnection class is used.
2. JarURLConnection-If however, we are trying to establish a connection to a jar file on the web, then Jar file on the web , then JarURLConnection is used.

Opening URLConnections

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

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

- You don't always perform all these steps. For instance, if the default setup for a particular kind of URL is acceptable, you can skip step 3. If you only want the data from the server and don't care about any meta information, or if the protocol doesn't provide any metainformation, 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.

1. Construct a URL object.

- Simply create a new `URL` object for a given URL like this:

```
URL url = new URL("http://www.google.com");
```

This constructor will throw a `MalformedURLException` if the URL is malformed. This checked exception is a subclass of `IOException`.

2. Obtain a URLConnection object from the URL

- A `URLConnection` instance is obtained by invoking the `openConnection()` method on the URL object:

```
URLConnection urlCon = url.openConnection();
```

- If the protocol is http://, you can cast the returned object to an `HttpURLConnection` object:

```
HttpURLConnection httpCon = (HttpURLConnection) url.openConnection();
```

Note that the `openConnection()` method doesn't establish an actual network connection. It just returns an instance of `URLConnection` class.

The network connection is made explicitly when the `connect()` method is invoked, or implicitly when reading header fields or getting an input stream / output stream.

The URL's `openConnection()` method throws `IOException` if an I/O error occurs.

3. Configure the URLConnection

Before actually establish the connection, you can configure various aspects that affect the ongoing communication between the client and the server, such as timeout, cache, HTTP request method, etc.

The `URLConnection` class provides the following methods for configuring the connection:

- **`setConnectTimeout`**`(int timeout)`: sets the connection timeout in milliseconds. A `java.net.SocketTimeoutException` is thrown if the timeout expires before the connection can be established. A timeout of zero indicates infinite timeout (the default value).
- **`setReadTimeout`**`(int timeout)`: sets the read timeout in milliseconds. After the timeout expires and there's no data available to read from the connection's input stream, a `SocketTimeoutException` is raised. A timeout of zero indicates infinite timeout (the default value).
- **`setDefaultUseCaches`**`(boolean default)`: sets whether the `URLConnection` uses caches by default or not (default is true). This method affects future instances of the `URLConnection` class.

3. Configure the URLConnection

- **`setUseCaches`**`(boolean useCaches)`: sets whether this connection uses cache or not (default is true).
- **`setDoInput`**`(boolean doInput)`: sets whether this `URLConnection` can be used for reading content from the server (default is true).
- **`setDoOutput`**`(boolean doOutput)`: sets whether this `URLConnection` can be used for sending data to the server (default is false).
- **`setIfModifiedSince`**`(long time)`: sets the last modified time of the content retrieved by the client, primarily for HTTP protocol. For example, if the server finds that the content has not changed since the specified time, it doesn't fetch the content and returns status code 304 - not modified. The client will get the fresh content if it has been modified more recently than the specified time.
- **`setAllowUserInteraction`**`(boolean allow)`: enables or disables user interaction, e.g. popping up an authentication dialog if required (default is false).
- **`setDefaultAllowUserInteraction`**`(boolean default)`: sets the default value for user interaction for all future `URLConnection` objects.
- **`setRequestProperty`**`(String key, String value)`: sets a general request property specified by a key=value pair. If a property with the key already exists, the old value is overwritten with the new value
- Note that these methods should be invoked before establishing the connection. Some methods throw `IllegalStateException` if the connection is already made.

3. Configure the URLConnection

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

Method	Description
protected boolean allowUserInteraction()	This method specifies whether user interaction is allowed. It is false by default. Since, this variable is protected, you can use the public getAllowUserInteraction() method to read its value, and the public setAllowUserInteraction () method to set it:
protected Boolean connected()	It specifies whether a communication link to a URL is established or not; true indicates it can be established; false indicates it cannot be established.

3. Configure the URLConnection

Method	Description
protected boolean doInput()	It specifies whether we can read from the specified URL resource or not, true indicates we can read (default value), false indicates we cannot read.
protected boolean doOutput()	It examines whether we can write to a URL resource or not; true indicates we can write. If false-we cannot write.
protected URL url()	It specifies the URL object from which the URLConnection object has been established.
protected boolean useCaches()	Caching means temporary storage of web documents in the local system. It indicates whether caching is allowed or not; true indicates it is allowed and false indicates the protocol always tries to get a fresh copy of a resource.
protected long ifModifiedSince	It returns the value of the object's ifModifiedSince field. If the document has changed since that time, the server will send it as usual. Otherwise , it will reply with a 304 Not Modified message.

4. Read the header fields

- Once the connection is made, the server processes the URL request and sends back a response that consists of metadata and the actual content. The metadata is a collection of key=value pairs which are called header fields.
- The header fields reveal information about the server, status code, protocol information, etc. The actual content can be in text, HTML, image, etc. depending on the type of the document.

4. Read the header fields

- The `URLConnection` class provides the following methods for reading the header fields:
- **`getHeaderFields()`**: returns a map that contains all header fields. The key is field name and the value is a list of String represents the corresponding field values.
- **`getHeaderField(int n)`**: reads the value of the n-th header field.
- **`getHeaderField(String name)`**: reads the value of the named header field.
- **`getHeaderFieldKey(int n)`**: reads the key of the n-th header field.
- **`getHeaderFieldDate(String name, long default)`**: reads the value of the named field parsed as `Date`. If the field is missing or the value is malformed, the default value is returned instead.
- **`getHeaderFieldInt(String name, int default)`**: reads the value of the named field parsed as an integer number. If the field is missing or the value is malformed, the default value is returned instead.
- **`getHeaderFieldLong(String name, long default)`**: reads the value of the named field parsed as a long number. If the field is missing or the value is malformed, the default value is returned instead.

4. Read the header fields

- These are general methods for reading any header fields. And for some frequently-accessed header fields, the `URLConnection` class provides more specific methods:
- **`getContentEncoding()`**: reads the value of the content-encoding header field, which indicates the encoding type of the content.
- **`getContentLength()`**: reads the value of the content-length header field, which indicates the size of the content (in bytes).
- **`getContentType()`**: reads the value of the content-type header field, which indicates the type of content.
- **`getDate()`**: reads the value of the Date header field, which indicates the date time on the server.
- **`getExpiration()`**: reads the expires header field value, indicating the time after which the response is considered stale. This is for cache control.
- **`getLastModified()`**: reads the value of the last-modified header field, which indicates the last modified time of the content.

And the subclass `HttpURLConnection` provides an additional method:

- **`getResponseCode()`**: returns the HTTP status code sent by the server.
- Note that when the header fields are read, the connection is implicitly established, without calling `connect()`.

5. Get an input stream and read data

- To read the actual content, you need to obtain an InputStream instance from the connection, and then use the InputStream's read() methods to read the data:

```
InputStream inputStream = urlCon.getInputStream();
byte[] data = new byte[1024];
inputStream.read(data);
```

The InputStream's read() is a low-level method that reads data to an array of bytes. So it is more convenient to wrap the InputStream in an InputStreamReader for reading data to characters:

```
InputStream inputStream = urlCon.getInputStream();
InputStreamReader reader = new InputStreamReader(inputStream);

int character = reader.read();  // reads a single character
char[] buffer = new char[2048];
reader.read(buffer);    // reads to an array of characters
```

5. Get an input stream and read data

Or wrap the `InputStream` in a `BufferedReader` for reading data to Strings:

```
BufferedReader reader = new BufferedReader(new InputStreamReader(input));
String line = reader.readLine(); // reads a line
```

- Note that the `getInputStream()` method can throw the following exceptions:
- `IOException`: if an I/O error occurs while creating the input stream.
- `SocketTimeoutException`: if the read timeout expires before data is available for read.
- `UnknownServiceExcepetion`: if the protocol does not support input.

6. Get an output stream and write data

To send data to the server, you have to enable output on the connection first:

```
urlCon.setDoOutput(true);
```

Then get the `OutputStream` object associated with the connection, and then use the `OutputStream`'s `write()` methods to write the data:

```
OutputStream outputStream = urlCon.getOutputStream();
byte[] data = new byte[1024];
outputStream.write(data);
```

As you can see, the `OutputStream`'s `write()` is a low-level method that writes an array of bytes. So it is more convenient to wrap the `OutputStream` in an `OutputStreamWriter` for writing characters

```
OutputStreamWriter writer = new OutputStreamWriter(outputStream);
int character = 'a';
writer.write(character); // writes a single character
char[] buffer = new char[2048];
writer.write(buffer); // writes an array of characters
```

6. Get an output stream and write data

Or wrap the `OutputStream` in a `PrintWriter` for writing Strings:

```
PrintWriter writer = new PrintWriter(outputStream);
String line = "This is String";
writer.print(line);
```

Note that the `getOutputStream()` method can throw `IOException` or `UnknownServiceException`.

7. Close the connection

To close the connection, invoke the `close()` method on either the `InputStream` or `OutputStream` object. Doing that may free the network resources associated with the `URLConnection` instance.

Differences between URL &URL Connection Class

- URL Connection provides access to the MIME header associated with an HTTP 1.0 response.
- URL Connection lets you configure the request parameters sent to the server.
- URL Connection lets you write data to the server as well as read data from the server.

Reading the Header

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

HTTP/1.1 301 Moved Permanently

Date: Sun, 21 Apr 2013 15:12:46 GMT

Server: Apache

Location: http://www.ibiblio.org/

Content-Length: 296

Connection: close

Content-Type: text/html; charset=iso-8859-1

Reading Data from a Server

- There’s a lot of information there. In general, an HTTP header may include the content type of the requested document, the length of the document in bytes, the character set in which the content is encoded, the date and time, the date the content expires, and the date the content was last modified. However, the information depends on the server; some servers send all this information for each request, others send some information, and a few don’t send anything. The methods of this section allow you to query a URL Connection to find out what metadata the server has provided.
- Aside from HTTP, very few protocols use MIME headers (and technically speaking, even the HTTP header isn’t actually a MIME header; it just looks a lot like one). When writing your own subclass of URLConnection, it is often necessary to override these methods so that they return sensible values. The most important piece of information you may be lacking is the content type. URLConnection provides some utility methods that guess the data’s content type based on its filename or the first few bytes of the data itself.

Retrieving Specific Header Fields

- The first six methods request specific, particularly common fields from the header. These are:
  - Content-type
  - Content-length
  - Content-encoding
  - Date
  - Last-modified
  - Expires

Retrieving Specific Header Fields

1.public String getContentType()

The getContentType() method returns the MIME content of the data. It throws no exceptions and returns null if the content type isn't available. text/html will be the most common content type you'll encounter when connecting to web servers. Other commonly used types include text/plain, image/gif, application/xml, and image/jpeg.

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

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

2.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. Many servers send Content-length headers only when they're transferring a binary file, not when transferring a text file.

Retrieving Specific Header Fields

3.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.

4.public long getDate()

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

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

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

Retrieving Specific Header Fields

5.public long getExpiration()

- Some documents have server-based expiration dates that indicate when the document should be deleted from the cache and reloaded from the server. getExpiration() is very similar to getDate(), differing only in how the return value is interpreted. It returns a long indicating the number of milliseconds after 12:00 A.M., GMT, January 1, 2021, 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.

6.Public long getLastModified()

- getLastModified method returns the date on which the document was last modified.Again, the date is given in miliseconds since midnight, GMT, January 1, 2021. If header does not include Lastmodified header, 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. If the requested header is found, it is returned. Otherwise, the method returns null.

Retrieving Arbitrary Header Fields

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 content-encoding = 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");

Retrieving Arbitrary Header Fields

2.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);
```

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.

Retrieving Arbitrary Header Fields

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 (e.g., the Expires, Date, or Last-modified headers). To convert the string to an integer, getHeaderFieldDate() uses the parseDate() method of java.util.Date. The parseDate() method does a decent job of understanding and converting most common date formats, but it can be stumped—for instance, if you ask for a header field that contains something other than a date. If parseDate() doesn't understand the date or if getHeaderFieldDate() is unable to find the requested header field, getHeaderFieldDate() returns the default argument. For example:

```
Date expires = new Date(uc.getHeaderFieldDate("expires", 0));

long lastModified = uc.getHeaderFieldDate("last-modified", 0);

Date now = new Date(uc.getHeaderFieldDate("date", 0));
```

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

Retrieving Arbitrary Header Fields

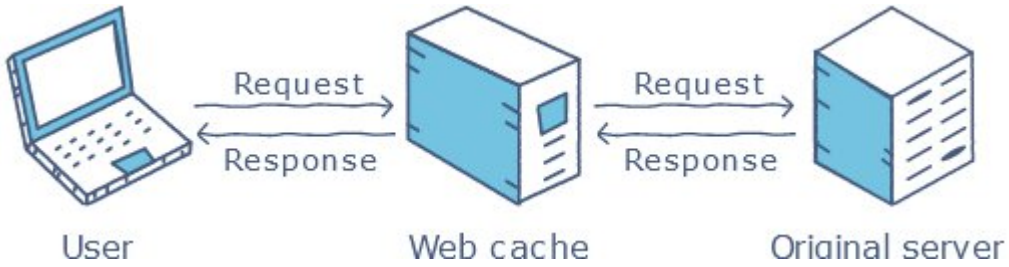
5.public int getHeaderFieldInt(String name, int default)

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

- **int** contentLength = uc.getHeaderFieldInt("content-length", -1);
- In this code fragment, getHeaderFieldInt() returns –1 if the Content-length header isn’t present.

Web Cache

The **web cache** stores recently requested objects in its memory. When requests for these objects are placed again, the cache (not the original server) services these HTTP requests. If the requested object is not available in the cache, then the web cache contacts the original server to obtain the object. The purpose of caching is to reduce response time by avoiding calls to the actual webserver.



Web Cache

The following steps demonstrate how web caching works

1. The user's browser makes an HTTP request for the desired object to the web cache.
2. If the web cache does not have this object in its memory, it requests the original server for the file, sends the object to the user's browser, and stores a copy of it along with the last-modified date of the object.
3. When another user requests the same object, the object is available in the cache. Before sending the object to the requesting user, the cache needs to ensure that this object has not been modified on the original server.
4. The web cache sends a conditional GET request to the original webserver, i.e., it tells the webserver to send the file *only* if it has been modified since the web cache's last-modified date.
5. If the object has not been modified at the original web server, then the response message will not contain the requested object, and the web cache will service the user's request using the object in its memory.

Cache-Control Header

Cache-control is an HTTP header used to specify browser caching (Browser caching involves a visitor's browser downloading your website's resources) policies in both client requests and server responses. Policies include how a resource is cached, where it's cached and its maximum age before expiring (i.e., time to live). Time to live is the time that an object is stored in a caching system before it's deleted or refreshed.

Cache-Control Header

The client is using the cache request directives in its HTTP request as follows:

Directive of Cache Request and Description
no-cache It indicates that the response which is returned and cannot be used to satisfy a subsequent request to the same URL
no-store It indicates that the client will not store anything about the request of the client or the response of the server.
max-age = seconds It is used to indicate that the client is willing to accept a response whose age is not greater than the time which is specified in seconds. For example, cache-control: max-age=120 means that the returned resource is valid for 120 seconds, after which the browser has to request a newer version.
only-if-cached It does not retrieve new data. A document can be send by the cache only if it is in the cache and should not contact the origin-server to see if it exists a newer copy.

Cache-Control Header

The directives of cache response can be used by the server in its HTTP response as follows:

Directive of Cache Response and Description
no-cache It indicates that the response which is returned cannot be used to satisfy a subsequent request to the same URL.
no-store It indicates that the client will not store anything about the request of the client or the response of the server.
max-age = seconds It is used to indicate that the client is willing to accept a response whose age is not greater than the time which is specified in seconds.
public It indicates that any cache may be cached the response.
private It is used to indicate that some part of the response message or all the response message is intended for a single user. It must not be cached by a shared cache.

Web Cache for java

There are three abstract classes in the java.net package. They are:

- **ResponseCache**
- **CacheRequest**
- **CacheResponse**

Web Cache for java

ResponseCache

The Java ResponseCache class represents the implementations of the URLConnection caches. It decides which resources should be cached, and for how long they should be cached .

Method	Description
get(URI uri, String rqstMethod, Map<String, List<String>> rqstHeaders)	This method retrieves the cached response based on the requesting uri, request method, and request headers
getDefault()	This method returns the system-wide response cache.
put(URI uri, URLConnection conn)	The protocol handler calls this method after a resource has been retrieved, and the ResponseCache decides whether to store the resource in its cache or not.
setDefault(ResponseCache responseCache)	This method sets or unsets the system-wide cache.

Web Cache for java

CacheRequest

CacheRequest class is used in java whenever there is a need for, storage of resources in ***ResponseCache***. More precisely instances of this class provide an advantage for the OutputStream object to store resource data into the cache, in fact, This OutputStream object is invoked by protocol handlers.

Two methods present in this class which are as follows:

1. **abort()** Method-It is used for aborting cache response, whenever an IOException occurs current cache operation is aborted. Hence, in simpler words, it aborts the attempt to cache the response.
   Syntax :
   public abstract void abort()

2. getBody() Method-It simply returns an InputStream from which the response body can be accessed.

Syntax:

```
public abstract OutputStream getBody ()
```

.

Web Cache for java

CacheResponse

CacheResponse is an abstract class that represents channels for retrieving resources from the ResponseCache. The objects of this class provide an InputStream that returns the entity-body and the associated response headers.

Methods: CacheResponse class provides two methods which are as follows:
1.getBody() method
2. getHeaders() method

Method 1: *getBody()* method returns an InputStream from which the response body can be accessed.
Syntax:
public abstract InputStream getBody() throws IOException

Return Type: This method returns the response body as an InputStream.

Method 2: *getHeaders()* method returns returns the headers stored.
Syntax:
public abstract Map<String,List<String>> getHeaders() throws IOException

Security Considerations

URLConnection objects are subject to all the usual security restrictions about making network connections, reading or writing files, and so forth. The Internet was virtually wide open, operating with a high level of trust and a low level of security. Now, because there are millions of users, security has become a huge concern. Companies are securing their networks to prevent uncontrolled access to their private networks from the outside.

Before attempting to connect a URL, you may want to know whether that connection will be allowed. Starting in Java 1.2, the

URLConnection class has a **getPermission( )** method:

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** will return different subclasses of **java.io.Permission.**

Permissions are generally used for two purposes: to protect caches of objects obtained through URLConnections, and to check the right of a recipient to learn about a particular URL. In the first case, the permission should be obtained *after* the object has been obtained. For example, in an HTTP connection, this will represent the permission to connect to the host from which the data was ultimately fetched. In the second case, the permission should be obtained and tested *before* connecting.

Proxies and Streaming Mode

Many users behind firewalls 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.

Write short note on:

Streaming Mode
Web Cache
UrlConnection class
Cookie
Cookie Store
Proxies
Url and url class
Network Interface Class
InetAddress
Teaching Reachability
Web Server Logfile
Write a program to download a page and read its properties.
How do you read data from server? Explain with an example.

- Which is not an instance field of URLConnection Class.

a) Connected

b) lastModified

c) useCaches

d) doInput

- URL can be identified into how many parts?

a) 5

b) 2

c) 3

d) 4

- Which of these methods is used to know when was the URL last modified?

a) LastModified()

b) getLastModified()

c) GetLastModified()

d) getLastModified()

- Which of these is a wrapper around everything associated with a reply from an http server?

a) HTTP()

b) HttpResponse()

c) HttpRequest()

d) HttpServer()

- What does the java.net.InetAddress class represent?
- Socket
- Ip Address
- Protocol
- MAC Address

- Which of these methods is used to know the type of content used in the URL ?

a) ContentType()

b) contentType()

c) GetContentType()

d) getContentType()