UNIT-5

Unit 5: URLConnections

5.1 Openning URLConnections
5.2 Reading Data from Server
5.3 Reading Header: Retrieving specific Header Fields and Retrieving Arbitrary Header Fields
5.4 Cache: Web Cache for Java
5.5 Configuring the Connection: protected URL url, protected boolean connected, protected boolean allowUserInteraction, protected boolean doInput, protected boolean doOutput, protected boolean ifModificationSince, protected boolean useCaches and Timeouts
5.6 Configuring the Client Request HTTP Header
5.7 Security Considerations for URLConnections
5.8 Guessing MIME Media Types
5.9 HttpURLConnection: The Request Methods, Disconnecting from the Server, Handling Server Responses, Proxies and Streaming Mode

URLconnection

- URLConnection is a Java class that allows you to connect to a URL and perform various operations, such as reading from or writing to the URL. It provides a convenient way to communicate with web servers and retrieve data from them.
- To use URLConnection, you need to first **create a URL object** that represents the URL you want to connect to. You can then call the **openConnection()** method on the URL object to get a URLConnection object that represents the connection to the URL.
- Once you have a URLConnection object, you can use its various methods to perform operations on the URL, such as:

1. Reading from the URL using the **getInputStream()** method
2. Writing to the URL using the **getOutputStream()** method
3. Getting the response code using the **getResponseCode()** method
4. Setting request headers using the **setRequestProperty()** method

Opening URLConnections

- URLConnections can be opened in Java using the **openConnection()** method of the URL class. This method returns a **URLConnection object** that represents the **connection to the URL**.

```
URL url = new URL(spec:"http://www.example.com");
URLConnection conn = url.openConnection();
```

Reading from Server

1. **Create a URL object**: Instantiate a URL object with the URL of the server you want to access.
2. **Open a connection to the server**: Call the **openConnection()** method on the URL object to create a URLConnection object.
3. **Configure the connection**: Set any properties on the URLConnection object that you need to, such as request headers or timeouts.
4. **Send the request**: Send the request to the server by calling the appropriate method on the URLConnection object, such as **getInputStream**() or **getResponseCode**().
5. **Read the response**: If the server returns data, you can read it from the InputStream returned by the URLConnection object. You can use a **BufferedReader** or other I/O classes to read the data.
6. **Close the connection**: Always close the InputStream and the URLConnection object when you're finished reading data from the server.

Reading from Server

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

public class DemoUrlConnection {
    Run | Debug
    public static void main(String[] args) throws Exception {
        URL url = new URL(spec:"http://www.example.com");
        URLConnection conn = url.openConnection();
        BufferedReader in = new BufferedReader(new InputStreamReader(conn.getInputStream()));

        String inputLine;
        while ((inputLine = in.readLine()) != null) {
            System.out.println(inputLine);
        }
        in.close();
    }
}
```

PROGRAM TO OBTAIN HTTP HEADER

```java
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.List;
import java.util.Map;

public class HeaderExample {
    Run | Debug
    public static void main(String[] args) throws Exception {
        URL url = new URL(spec:"http://www.example.com");
        HttpURLConnection conn = (HttpURLConnection) url.openConnection()
        conn.setRequestMethod(method:"GET");

        Map<String, List<String>> headers = conn.getHeaderFields();
        for (String key : headers.keySet()) {
            System.out.println(key + ": " + headers.get(key));
        }
    }
}
```

```
null: [HTTP/1.1 200 OK]
X-Cache: [HIT]
Server: [ECS (oxr/8374)]
Last-Modified: [Thu, 17 Oct 2019 07:18:26 GMT]
Date: [Mon, 10 Apr 2023 12:47:57 GMT]
Accept-Ranges: [bytes]
Cache-Control: [max-age=604800]
Etag: ["3147526947"]
Vary: [Accept-Encoding]
Expires: [Mon, 17 Apr 2023 12:47:57 GMT]
Content-Length: [1256]
Age: [259338]
Content-Type: [text/html; charset=UTF-8]
```

ARBITRARY HEADER FIELD

Methods of the URLConnection class in Java

1. getContentType(): This method returns a String representing the MIME type of the resource, such as "text/html" for an HTML document.
2. getContentEncoding(): This method returns a String representing the encoding used to encode the resource's content, such as "gzip" for a resource that has been compressed with the gzip algorithm.
3. getDate(): This method returns the date and time when the resource was last modified, as a long value representing the number of milliseconds since January 1, 1970, 00:00:00 GMT.
4. getExpiration(): This method returns the date and time when the resource is set to expire, as a long value representing the number of milliseconds since January 1, 1970, 00:00:00 GMT.
5. getContentLength(): This method returns the size of the resource's content, in bytes, as a long value.

ARBITRARY HEADER FIELD

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

public class ArbitaryHeader {
    Run | Debug
    public static void main(String[] args) throws IOException {
        String urlStr = "https://example.com";
        URL url = new URL(urlStr);
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        System.out.println("Content-Type::" + conn.getContentType());
        System.out.println("Content-Length::" + conn.getContentLength());
        System.out.println("Expiration-Date::" + conn.getExpiration());
        System.out.println("Last-Modified::" + conn.getLastModified());
        System.out.println("Date::" + conn.getDate());
        System.out.println("Content-encoding::" + conn.getContentEncoding());

    }
}
```

```
Content-Type::text/html; charset=UTF-8
Content-Length::1256
Expiration-Date::1681746065000
Last-Modified::1571296706000
Date::1681141265000
Content-encoding::null
```

CACHES : WEB CACHE FOR JAVA

- Web Cache for Java By default, **Java does not cache anything** However, you can implement a caching mechanism by using a subclass of the **ResponseCache** class, which is provided by **the java.net** package. The **ResponseCache** class provides a way to cache **HTTP and HTTPS responses**, and you can implement your own **cache by extending this class and implementing** the get and put methods. To implement a cache for the URL class in Java, you would typically follow these steps:

1. Create a **concrete subclass of ResponseCache** that implements the get and put methods. The get method should return a **CacheResponse object** if a cached response exists, or null if there is no cached response. The put method should store a response in the cache for a given URL and request method.
2. Create a **concrete subclass of CacheRequest** that represents a request that is being cached. This class should implement the **getBody** method, which returns an **OutputStream** that can be used to write the body of the request to the cache.
3. Create a **concrete subclass of CacheResponse** that represents a response that is being cached. This class should implement the **getBody** method, which returns an **InputStream** that can be used to read the body of the response from the cache.
4. Use the **ResponseCache.setDefault** method to set the default **ResponseCache** for the Java application to your concrete subclass of ResponseCache.

TO CONFIGURE A URLCONNECTION

- To configure a **URLConnection instance**, you can use the various methods provided by the class. Here are some commonly used methods:

1. **setRequestMethod(String method):** sets the request method (e.g., GET or POST)
2. **setRequestProperty(String key, String value):** sets a request header
3. **setDoOutput(boolean dooutput):** sets whether output (e.g., writing to the server) is allowed for this connection
4. **setDoInput(boolean doinput):** sets whether input (e.g., reading from the server) is allowed for this connection
5. **setConnectTimeout(int timeout):** sets the maximum time to wait for a connection to be established, in milliseconds
6. **setReadTimeout(int timeout):** sets the maximum time to wait for data to be read, in milliseconds

TO CONFIGURE A URLCONNECTION

```java
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;

public class URLConfExample {
    Run | Debug
    public static void main(String[] args) throws Exception {
        URL url = new URL(spec:"http://example.com");
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setRequestMethod(method:"GET");
        conn.setRequestProperty(key:"Accept", value:"text/html");
        conn.setDoOutput(dooutput:false);
        conn.setDoInput(doinput:true);
        conn.setConnectTimeout(timeout:5000);
        conn.setReadTimeout(timeout:5000);
        conn.connect();
        BufferedReader in = new BufferedReader(
                new InputStreamReader(
                    conn.getInputStream()));

        String line;
        while ((line = in.readLine()) != null) {
            System.out.println(line);
        }
        in.close();
    }
}
```

URLCONNECTIONS

1. **url**: This is a **java.net.URL** object that represents the URL that this instance of the class is connected to.
2. **connected**: This is a boolean value that represents whether or not the connection to the URL has been established.
3. **allowUserInteraction**: This is a boolean value that indicates whether or not the user can interact with the URL connection.
4. **doInput**: This is a boolean value that indicates whether or not this URL connection allows input.
5. **doOutput**: This is a boolean value that indicates whether or not this URL connection allows output.
6. **ifModifiedSince**: This is a long value that represents the date and time since the last modification of the resource that the URL points to. If the resource has not been modified since the specified date and time, the server can send a "Not Modified" response instead of the full content of the resource.
7. **useCaches**: This is a boolean value that indicates whether or not the URL connection should use the caching mechanism.

URLCONNECTIONS

```java
import java.io.*;
import java.net.*;
public class SomeUrlMethod {
    Run | Debug
    public static void main(String[] args) throws Exception {
        // Create a URL object
        URL url = new URL(spec:"https://www.example.com");
        // Open a connection to the URL
        URLConnection connection = url.openConnection();
        // Set some properties of the connection
        connection.setConnectTimeout(timeout:5000);
        connection.setReadTimeout(timeout:10000);
        connection.setDoInput(doinput:true);
        connection.setDoOutput(dooutput:false);
        // Print out some information about the connection
        System.out.println("Content Type: " + connection.getContentType());
        System.out.println("Content Length: " + connection.getContentLength());
        System.out.println("Allow User Interaction: " + connection.getAllowUserInteraction());
        System.out.println("Do Input: " + connection.getDoInput());
        System.out.println("Do Output: " + connection.getDoOutput());
        System.out.println("If Modified Since: " + connection.getIfModifiedSince());
        // Get the input stream from the connection and read the response
        BufferedReader in = new BufferedReader(new InputStreamReader(connection.getInputStream()));
        String inputLine;
        while ((inputLine = in.readLine()) != null) {
            System.out.println(inputLine);
        }
        in.close();
    }
}
```

CONFIGURING THE CLIENT REQUEST HTTP HEADER

To configure the client request HTTP header, you need to include the appropriate headers in the request. HTTP headers are additional pieces of information sent along with the request or response in the form of key-value pairs. Some commonly used headers are:

- User-Agent: specifies the user agent string of the client making the request. This helps the server identify the client's operating system, browser, etc.
- Accept: specifies the MIME types of content that the client can understand.
- Authorization: specifies the credentials to be used for accessing the resource.
- Content-Type: specifies the MIME type of the content being sent in the request body.
- Cache-Control: specifies how caching should be done for the request and/or response.

CONFIGURING THE CLIENT REQUEST HTTP HEADER

HTTPRequestExample.java > ...

```java
import java.net.HttpURLConnection;
import java.net.URL;

public class HTTPRequestExample {
    Run | Debug
    public static void main(String[] args) throws IOException {
        URL url = new URL(spec:"https://mechicampus.edu.np");
        HttpURLConnection con = (HttpURLConnection) url.openConnection();

        con.setRequestMethod(method:"GET");
        con.setRequestProperty(key:"User-Agent", value:"Mozilla/5.0");
        con.setRequestProperty(key:"Accept", value:"application/json");
        con.setRequestProperty(key:"Authorization", value:"Mechi <2543345345>");
        con.setRequestProperty(key:"Content-Type", value:"application/json");
        con.setRequestProperty(key:"Cache-Control", value:"no-cache");

        int status = con.getResponseCode();
        // int status = con.getResponseCode();
        System.out.println("Response status: " + status);
    }
}
```

SECURITY CONSIDERATIONS FOR URLCONNECTIONS

- Object of **URLConnection** class are subject to all the usual **security restrictions about making network connections.** The **getPermission**() is a method in the **URLConnection** class in Java that returns the **Permission object representing the permission necessary to make the connection.**
- This method is used to **check if the calling code has the necessary permissions to establish a connection** with the URL. If the permission is not granted, an IOException 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:

```java
import java.net.*;
import java.security.Permission;
import java.io.*;
public class URLPermissionExample {
    Run | Debug
    public static void main(String[] args) {
        try {
            // Create URL object
            URL url = new URL(spec:"https://www.example.com");
            // Open connection to URL
            URLConnection connection = url.openConnection();
            // Check if the calling code has permission to connect
            Permission permission = connection.getPermission();
            if (permission != null) {
                System.out.println("Permission granted: " + permission.getName());
            } else {
                System.out.println(x:"No permission required to connect");
            }
        } catch (IOException e) {
            System.err.println("IOException: " + e.getMessage());
        }
    }
}
```

GUESSING MIME MEDIA TYPES

The URLConnection class provides two static methods to help programs **figure out the MIME type** of some data
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.

```
import java.net.URL;
import java.net.URLConnection;
import java.io.IOException;
public class GuessMediaTypes {
    Run | Debug
    public static void main(String[] args) {
        try {
            URL url = new URL(spec:"https://mechicampus.edu.np/wp-content/uploads/2020/01/BBA-lesson-plan.docx");
            URLConnection connection = url.openConnection();
            String contentType = URLConnection.guessContentTypeFromName(url.getFile());
            System.out.println("MIME type: " + contentType);
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

```
MIME type: application/vnd.openxmlformats-officedocument.wordprocessingml.document
```

HTTPURLCONNECTION:THE REQUEST METHOD

- in **HttpURLConnection**, the request method is the **HTTP method** used to send the request to the server. The most commonly used request methods are **GET, POST, PUT, DELETE, HEAD, and OPTIONS**.
- To set the request method in HttpURLConnection, you can use the **setRequestMethod()** method.

```java
import java.net.HttpURLConnection;
import java.net.URL;
import java.io.BufferedReader;
import java.io.InputStreamReader;
public class SetRequestMethod {
    Run | Debug
    public static void main(String[] args) {
        try {
            URL url = new URL(spec:"https://www.example.com");
            HttpURLConnection connection = (HttpURLConnection) url.openConnection();
            connection.setRequestMethod(method:"POST");
            // Set other request headers and parameters here
            BufferedReader in = new BufferedReader(new InputStreamReader(
                                    connection.getInputStream()));
            String inputLine;
            StringBuffer content = new StringBuffer();
            while ((inputLine = in.readLine()) != null) {
                content.append(inputLine);
            }
            in.close();
            connection.disconnect();
            System.out.println("Response: " + content.toString());
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

HANDLING SERVER RESPONSES, DISCONNECTING FROM THE SERVER

- In Java, you can disconnect from a server using the **disconnect() method** of HttpURLConnection. This method releases any resources associated with the connection and allows them to be reused by other connections.
- **The getResponseCode()** method of HttpURLConnection returns the HTTP response code returned by the server in response to the request. This **code indicates the status of the request**, and can be used to determine if the request was **successful** or if there was an **error**.

```java
import java.net.HttpURLConnection;
import java.net.URL;
import java.io.BufferedReader;
import java.io.InputStreamReader;
public class SetRequestMethod {
    Run | Debug
    public static void main(String[] args) {
        try {
            URL url = new URL(spec:"https://www.example.com");
            HttpURLConnection connection = (HttpURLConnection) url.openConnection();
            connection.setRequestMethod(method:"POST");
            // Set other request headers and parameters here
            BufferedReader in = new BufferedReader(new InputStreamReader(
                                    connection.getInputStream()));
            String inputLine;
            StringBuffer content = new StringBuffer();
            int responseCode = connection.getResponseCode();
            System.out.println("Response code: " + responseCode);
            while ((inputLine = in.readLine()) != null) {
                content.append(inputLine);
            }
            in.close();
            connection.disconnect();
            System.out.println("Response: " + content.toString());
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

PROXY: UsingProxy() METHOD

- In computer networking, a proxy is an **intermediary server** between clients and other servers. The **usingProxy**() method is a method of the **URLConnection** class in Java that returns a boolean value indicating **whether the URL connection is using a proxy server.**
- Note that if the connection is using a proxy server, you may need to **set additional proxy server settings such** as the **host name** and **port number**, as well as any authentication credentials required to access the proxy server. You can do this using the **System.setProperty()** method or by using a Proxy object to configure the proxy settings for the connection.

```
import java.net.HttpURLConnection;
import java.net.URL;
public class UsingProxy {
    Run | Debug
    public static void main(String[] args) throws Exception {
        URL url = new URL(spec:"http://www.example.com");
        HttpURLConnection connection = (HttpURLConnection) url.openConnection();

        boolean usingProxy = connection.usingProxy();
        if (usingProxy) {
            System.out.println(x:"Using a proxy server");
        } else {
            System.out.println(x:"Not using a proxy server");
        }
    }
}
```

STREAMING MODE

- The **HttpURLConnection** class in Java provides three methods to set the **streaming mode for the output content** of an **HTTP** request:
- **setChunkedStreamingMode**(int chunkLength): This method enables chunked transfer encoding for the request and sets the size of each chunk. Chunked transfer encoding is a streaming method that allows **data to be sent in a series of chunks of a variable size**, rather than as a single, fixed-length block. This is useful for **streaming large or unknown-size data sets**, since it allows the **data to be sent in smaller, more manageable pieces**.
- **setFixedLengthStreamingMode**(int contentLength): This method **sets the content length of the request to a fixed value**. This is useful when the **size of the data to be sent is known in advance**.
- **setFixedLengthStreamingMode**(long contentLength): This method is similar to the previous method, but accepts a long value for the content length, allowing it to handle **larger data sets**.

```
HttpURLConnection conn = (HttpURLConnection) url.openConnection();
conn.setDoOutput(true);
conn.setRequestMethod("POST");
conn.setChunkedStreamingMode(1024); // send data in 1KB chunks
```

```
HttpURLConnection conn = (HttpURLConnection) url.openConnection();
conn.setDoOutput(true);
conn.setRequestMethod("POST");
int contentLength = data.getBytes().length;
conn.setFixedLengthStreamingMode(contentLength); // send data in a
```

```
HttpURLConnection conn = (HttpURLConnection) url.openConnection();
conn.setDoOutput(true);
conn.setRequestMethod("POST");
long contentLength = file.length();
conn.setFixedLengthStreamingMode(contentLength); // send data in a
```