**Unit 3**

**URL**

**What is the URL?**

A URL is a type of uniform resource identifier and is address of a resource on the World Wide Web and the protocol used to access it. It is used to indicate the location of a web resource to access the web pages. For example, to visit the google website, you will go to the URL www.google.com, which is the URL for the google website. The URL sends users to a specific resource online such as video, webpage, or other resources. When you search any query on Google, it will display the multiple URLs of the resource that are all related to your search query. The displayed URLs are the hyperlink to access the webpages.

**Different components of a URL explained with examples:**

1. Protocol: The protocol is the method used to access the resource. It specifies the rules for how the data is transferred over the network. The most common protocol is HTTP (Hypertext Transfer Protocol), but there are also other protocols like HTTPS, FTP, and SMTP. Example: [http://www.example.com](http://www.example.com/) or [https://www.example.com](https://www.example.com/)
2. Domain name: The domain name identifies the website or server where the resource is located. It usually consists of two or more parts separated by dots, such as "google.com" or "wikipedia.org". Example: [https://www.google.com](https://www.google.com/) or [https://www.wikipedia.org](https://www.wikipedia.org/)
3. Port number: The port number is an optional component that specifies the network port to be used for the request. If no port number is specified, the default port number for the protocol is used (e.g., 80 for HTTP, 443 for HTTPS). Example: [http://www.example.com:8080](http://www.example.com:8080/) or [https://www.example.com:8443](https://www.example.com:8443/)
4. Path: The path identifies the location of the resource on the server. It typically includes the directory structure and file name, such as "/index.html" or "/images/logo.png". Example: <https://www.example.com/products/phones/iphone12> or <https://www.example.com/about>
5. Query string: The query string is an optional component that is used to pass data to the server. It is a string of key-value pairs separated by "&" and preceded by a "?", such as "?search=apple&category=fruits".Example: <https://www.example.com/search?q=apple>or <https://www.example.com/products/phones?brand=apple&color=red>
6. Fragment identifier: The fragment identifier is an optional component that identifies a specific section within the resource. It is indicated by a "#" followed by a fragment identifier name, such as "#section2". Example: <https://www.example.com/about#team> or <https://www.example.com/products/phones#reviews>.

**Java 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. It is part of the java.net package and provides methods to retrieve information about a URL such as its protocol, host name, port number, path, query, and fragment components. The URL class can be used to establish a connection to a remote server or to retrieve data from a remote resource. It can also be used to encode or decode URLs and to compare URLs for equality. The URL class is useful for tasks such as web scraping, data mining, and web service integration in Java programs.

**The URL class in Java networking has several uses, including:**

1. Retrieving information about a URL: The URL class provides methods to retrieve information about a URL such as its protocol, host name, port number, path, query, and fragment components. This information can be used to establish a connection to a remote server or to retrieve data from a remote resource.
2. Establishing a connection to a remote server: The URL class can be used to establish a connection to a remote server using the openConnection() method. This method returns a URLConnection object that can be used to read from or write to the server.
3. Retrieving data from a remote resource: The URL class can be used to retrieve data from a remote resource using the openStream() method. This method returns an InputStream object that can be used to read data from the remote resource.
4. Encoding and decoding URLs: The URL class provides methods to encode or decode URLs using the URLEncoder and URLDecoder classes. These methods are useful for creating valid URLs that can be sent over the network.
5. Comparing URLs for equality: The URL class provides methods to compare URLs for equality using the equals() and sameFile() methods. These methods are useful for comparing URLs and determining if they refer to the same resource.
6. Parsing URLs: The URL class provides a convenient way to parse URLs using the URL constructor. This constructor takes a string representation of a URL and parses it into its component parts, allowing developers to work with URLs in a structured way.

Overall, the URL class is a powerful tool for working with URLs in Java networking, and it can be used for a wide range of tasks such as web scraping, data mining, and web service integration.

**Methods of URL Class**

The URL class in Java provides a variety of methods to work with URLs. Here are some of the most commonly used methods:

1. getProtocol(): returns the protocol of the URL, such as "http" or "https".
2. getHost(): returns the host name of the URL, such as "[www.example.com](http://www.example.com/)".
3. getPort(): returns the port number of the URL, or -1 if no port number is specified.
4. getPath(): returns the path component of the URL, such as "/path/to/file.html".
5. getQuery(): returns the query component of the URL, such as "key1=value1&key2=value2".
6. getRef(): returns the fragment component of the URL, such as "section1".
7. openConnection(): establishes a connection to the remote server and returns a URLConnection object.
8. openStream(): opens a connection to the remote server and returns an InputStream object to read data from the server.
9. toString(): returns a string representation of the URL.
10. toURI(): converts the URL to a URI object.
11. equals(Object obj): compares the URL with the specified object for equality.
12. sameFile(URL other): compares the URL with the specified URL for equality, ignoring the fragment component.

***import*** *java.net.\*;*

***public******class*** *URLExample {*

***public******static******void*** *main(String[] args) {*

***try*** *{*

*// create a URL object for the website we want to access*

*URL url =* ***new*** *URL("https://www.google.com");*

*// display the protocol of the URL (should be "https")*

*System.****out****.println("Protocol: " + url.getProtocol());*

*// display the host name of the URL (should be "www.example.com")*

*System.****out****.println("Host Name: " + url.getHost());*

*// display the port number of the URL (should be -1 since no port is specified)*

*System.****out****.println("Port Number: " + url.getPort());*

*// display the path component of the URL (should be an empty string since no path is specified)*

*System.****out****.println("Path: " + url.getPath());*

*// display the query component of the URL (should be null since no query is specified)*

*System.****out****.println("Query: " + url.getQuery());*

*// display the fragment component of the URL (should be null since no fragment is specified)*

*System.****out****.println("Fragment: " + url.getRef());*

*}* ***catch*** *(MalformedURLException e) {*

*System.****out****.println("Invalid URL");*

*}*

*}*

*}*

**Creating New URLs and its uses**

communicate with a web resource or web service over the internet. A URL (Uniform Resource Locator) is a standardized way to identify a resource on the internet, such as a webpage, an image file, or a RESTful API endpoint.

Here are some common use cases for creating a new URL in Java programming:

1. Downloading data: You may need to download data from a web resource, such as a webpage or JSON API endpoint. In this case, you can create a new URL object that points to the resource, open a connection to it, and read the data using input/output classes in Java.
2. Uploading data: Conversely, you may need to upload data to a web resource, such as a file or form data. In this case, you can create a new URL object that points to the resource, open a connection to it, and write the data using output classes in Java.
3. Validating URLs: You may need to validate that a URL is well-formed and points to a valid resource. In this case, you can create a new URL object and check for exceptions, such as MalformedURLException or IOException, when attempting to open a connection to it.
4. Testing connectivity: You may need to test connectivity to a web resource or server, such as in a health check or status monitoring system. In this case, you can create a new URL object that points to a resource on the server and check for exceptions when attempting to open a connection to it.

*import java.net.URL;*

*import java.net.HttpURLConnection;*

*import java.io.InputStream;*

*import java.io.BufferedReader;*

*import java.io.InputStreamReader;*

*import java.io.IOException;*

*public class URLDemo {*

*public static void main(String[] args) {*

*try {*

*// Create a new URL object*

*URL url = new URL("https://www.example.com");*

*// Open a connection to the URL*

*HttpURLConnection connection = (HttpURLConnection) url.openConnection();*

*// Set request properties*

*connection.setRequestMethod("GET");*

*connection.setRequestProperty("User-Agent", "Mozilla/5.0");*

*connection.setConnectTimeout(5000);*

*// Connect to the URL*

*connection.connect();*

*// Read data from the URL*

*InputStream in = connection.getInputStream();*

*BufferedReader reader = new BufferedReader(new InputStreamReader(in));*

*String line;*

*while ((line = reader.readLine()) != null) {*

*System.out.println(line);*

*}*

*in.close();*

*} catch (IOException e) {*

*e.printStackTrace();*

*}*

*}*

*}*

**Splitting**

Splitting in network programming refers to the process of dividing a large message or data set into smaller segments, known as packets or frames, for efficient transmission over a network. This approach is commonly used to improve network performance, reduce latency, and increase reliability by allowing data to be sent in smaller, more manageable pieces. Additionally, splitting enables the network to recover from transmission errors by retransmitting only the affected packets, rather than resending the entire message. Splitting can be implemented at different levels of the network protocol stack, and it is a key technique in modern network communication, particularly in applications that transmit large amounts of data.

**Splitting a URL into pieces can serve several purposes:**

1. Parsing: A URL is a structured string that follows a specific format, and splitting it into its individual components (such as the protocol, domain name, path, query string, etc.) can make it easier to parse and manipulate the different parts of the URL.
2. Validation: Splitting a URL can also help to validate it and ensure that it conforms to the expected format. For example, you can check whether the URL starts with a valid protocol (e.g., "http", "https", "ftp"), whether the domain name is valid and has a valid top-level domain (TLD), etc.
3. Modification: Splitting a URL can also make it easier to modify certain parts of the URL. For example, you can easily change the protocol or domain name of a URL without affecting the other parts.
4. Security: Splitting a URL can also help to detect and prevent certain security vulnerabilities, such as URL injection attacks. By separating the different parts of the URL and validating each part separately, you can ensure that the URL is safe and does not contain any malicious code or unexpected parameters.

*import java.net.URL;*

*public class URLSplitter {*

*public static void main(String[] args) throws Exception {*

*String urlString = "https://www.example.com/path/to/file.html?query1=value1&query2=value2#fragment";*

*URL url = new URL(urlString);*

*String protocol = url.getProtocol();*

*String host = url.getHost();*

*int port = url.getPort();*

*String path = url.getPath();*

*String query = url.getQuery();*

*String fragment = url.getRef();*

*System.out.println("Protocol: " + protocol);*

*System.out.println("Host: " + host);*

*System.out.println("Port: " + port);*

*System.out.println("Path: " + path);*

*System.out.println("Query: " + query);*

*System.out.println("Fragment: " + fragment);*

*}*

*}*

**Steps to Retrieving Data From URL**

To retrieve data from a URL using network programming, you can follow the steps below:

1. Establish a connection to the web server: Use the appropriate network protocol, such as HTTP or HTTPS, to connect to the server where the URL is located. You can use libraries like urllib or requests in Python to make this connection.
2. Send a request for data: After establishing the connection, send an HTTP request to the server for the data you want to retrieve. This request typically includes the HTTP method (GET, POST, PUT, etc.), headers, and any parameters required by the API.
3. Receive the response: The server will respond to your request with an HTTP response containing the requested data. The response may include headers, a status code, and a response body.
4. Parse the response: Depending on the format of the data you requested, you may need to parse the response to extract the relevant information. This can be done using libraries such as JSON, BeautifulSoup or XML parsers.
5. Close the connection: Once you have received the data, close the connection to the server to free up system resources and avoid potential security issues.

*import java.io.BufferedReader;*

*import java.io.InputStreamReader;*

*import java.net.URL;*

*import java.net.HttpURLConnection;*

*public class URLReader {*

*public static void main(String[] args) throws Exception {*

*// Replace the URL below with the actual URL you want to retrieve data from*

*URL url = new URL("https://example.com");*

*// Open a connection to the URL*

*HttpURLConnection conn = (HttpURLConnection) url.openConnection();*

*// Set the HTTP method to GET*

*conn.setRequestMethod("GET");*

*// Get the response code from the server*

*int responseCode = conn.getResponseCode();*

*// If the response code indicates success (200), read the data from the URL*

*if (responseCode == 200) {*

*BufferedReader in = new BufferedReader(*

*new InputStreamReader(conn.getInputStream()));*

*String inputLine;*

*StringBuffer response = new StringBuffer();*

*while ((inputLine = in.readLine()) != null) {*

*response.append(inputLine);*

*}*

*in.close();*

*// Print the data retrieved from the URL*

*System.out.println(response.toString());*

*} else {*

*System.out.println("Error: Could not retrieve data from URL.");*

*}*

*// Close the connection to the URL*

*conn.disconnect();*

*}*

*}*

**Utility Method**

In network programming, a utility method is a function or subroutine that provides a specific and useful functionality to help perform common tasks or operations related to network communication. These methods can include functions for sending or receiving data packets, handling errors, parsing data, establishing and maintaining connections, and more. Utility methods help simplify the programming process by providing reusable code and reducing the amount of code that needs to be written. They can also improve the overall performance and efficiency of network applications by providing optimized and standardized methods for handling common network tasks.

Utility methods are important in network programming because they provide a standardized and efficient way to perform common tasks related to network communication. Here are some of the key uses and benefits of utility methods:

1. Code reusability: Utility methods can be reused across different network applications, reducing the amount of code that needs to be written and increasing code efficiency.
2. Standardization: Utility methods provide a standardized way to perform common network tasks, making it easier for developers to understand and maintain network applications.
3. Error handling: Utility methods can be designed to handle common errors that occur during network communication, making it easier to identify and resolve issues.
4. Performance optimization: Utility methods can be optimized for performance, reducing the amount of processing time required to complete common network tasks.
5. Abstraction: Utility methods can provide a higher level of abstraction, allowing developers to focus on the overall functionality of their network applications rather than low-level details of network communication.

*import java.io.\*;*

*import java.net.\*;*

*public class NetworkUtilityExample {*

*// This is a utility method that establishes a connection between a client and server.*

*public static void establishConnection(String serverIP, int serverPort) throws IOException {*

*// Create a new Socket object and connect it to the server at the specified IP address and port number.*

*Socket socket = new Socket(serverIP, serverPort);*

*// Print a message indicating that the connection has been established.*

*System.out.println("Connection established with server at " + serverIP + ":" + serverPort);*

*// Close the socket to end the connection.*

*socket.close();*

*// Print a message indicating that the connection has been closed.*

*System.out.println("Connection closed");*

*}*

*public static void main(String[] args) {*

*// Specify the IP address and port number of the server.*

*String serverIP = "192.168.1.1";*

*int serverPort = 8080;*

*try {*

*// Call the utility method to establish a connection with the server.*

*establishConnection(serverIP, serverPort);*

*} catch (IOException e) {*

*// If an exception occurs, print an error message.*

*System.err.println("Error: " + e.getMessage());*

*}*

*}*

*}*

**URI**

A URI, or Uniform Resource Identifier, is a string of characters that identifies a resource on the Internet. It consists of two parts: the scheme, which identifies the protocol used to access the resource, and the identifier, which is the actual string of characters that identifies the resource. URIs are used to enable communication between different systems on the Internet, and can be either absolute or relative. They are used in many different contexts, such as in hyperlinks on web pages, in email messages, or in web services. URI is a standard defined by the Internet Engineering Task Force (IETF) in RFC 3986, and there are several different types of URIs, including URLs and URNs.

1. A URI is a string of characters that identifies a resource on the Internet. The resource can be a web page, an image, a file, or any other type of content.
2. URIs consist of two parts: the scheme and the identifier. The scheme identifies the protocol used to access the resource, such as HTTP or FTP. The identifier is the actual string of characters that identify the resource.
3. URIs are used to enable access to resources on the Internet, and can be used in many different contexts. For example, they can be used in hyperlinks on web pages, in email messages, or in web services.
4. URIs can be either absolute or relative. An absolute URI contains the full path to the resource, including the scheme, while a relative URI specifies the path relative to the current resource.
5. URIs are used to enable communication between different systems on the Internet. For example, when a web browser requests a web page, it sends a URI to the web server to identify the page to be retrieved.
6. URI is a standard defined by the Internet Engineering Task Force (IETF) in RFC 3986. This standard defines the syntax and semantics of URIs and provides guidelines for their use.
7. There are several different types of URIs, including Uniform Resource Locators (URLs), which are used to locate resources on the web, and Uniform Resource Names (URNs), which are used to identify resources by name rather than location.

**Process of Constructing URI**

1. Choose a scheme: The first step in constructing a URI is to choose the appropriate scheme for the type of resource being identified. The scheme identifies the protocol used to access the resource, such as HTTP or FTP.
2. Identify the resource: Once the scheme has been chosen, the next step is to identify the specific resource being accessed. This can be done using a combination of domain names, directory paths, and file names, depending on the type of resource.
3. Encode special characters: Certain characters, such as spaces and symbols, are not allowed in a URI and must be encoded using percent encoding. For example, a space would be replaced with %20.
4. Use relative or absolute paths: Depending on the context, a URI can use either a relative or an absolute path to identify the resource. A relative path is used when the resource is located relative to the current resource, while an absolute path specifies the full path to the resource.
5. Ensure uniqueness: A URI should be unique so that it can be used to identify a specific resource without confusion. This can be achieved by including unique identifiers, such as timestamps or random numbers, in the URI.
6. Consider readability: Although not required, it is generally a good practice to make a URI as readable as possible. This can be done by using descriptive names for resources and avoiding unnecessary complexity.
7. Validate the URI: Once the URI has been constructed, it should be validated to ensure that it is a valid and usable identifier for the resource. This can be done using a URI validator tool or by manually checking the syntax and structure of the URI.

*import java.net.URI;*

*public class ConstructURI {*

*public static void main(String[] args) {*

*try {*

*// Choose a scheme*

*String scheme = "http";*

*// Identify the resource*

*String domain = "www.example.com";*

*String path = "/example.html";*

*String resource = domain + path;*

*// Encode special characters*

*String encodedResource = URI.create(resource).toASCIIString();*

*// Use absolute path*

*URI uri = new URI(scheme, null, encodedResource, null);*

*// Ensure uniqueness (optional)*

*String uniqueURI = uri.toString() + "?timestamp=" + System.currentTimeMillis();*

*// Print the URI*

*System.out.println("URI: " + uniqueURI);*

*} catch (Exception e) {*

*e.printStackTrace();*

*}*

*}*

*}*

**Parts of URI**

A URI (Uniform Resource Identifier) consists of several parts that identify a resource on the internet. Here are the main parts of a URI along with examples:

1. Scheme: A scheme identifies the protocol to be used for accessing the resource. Examples include "http", "https", "ftp", "file", "mailto", etc. For instance, the URI "[http://www.example.com](http://www.example.com/)" uses the "http" scheme.
2. Authority: The authority component is used to specify the location of the resource. It typically includes a domain name or IP address and an optional port number. For example, in the URI "[http://www.example.com:8080](http://www.example.com:8080/)", "[www.example.com](http://www.example.com/)" is the domain name and "8080" is the port number.
3. Path: The path component identifies the specific resource being accessed within the domain. It starts with a slash character "/" and can include additional subdirectories or files. For example, in the URI "<http://www.example.com/path/to/resource.html>", "/path/to/resource.html" is the path component.
4. Query: The query component provides additional data to be passed to the server along with the request. It consists of a series of name-value pairs separated by "&" symbols, and starts with a question mark "?". For example, in the URI "<http://www.example.com/search?q=example>", "q=example" is the query component.
5. Fragment: The fragment component specifies a specific location within the resource being accessed. It starts with a hash symbol "#" and can include additional characters to identify a specific section of the resource. For example, in the URI "<http://www.example.com/resource.html#section3>", "section3" is the fragment component.

**Resolving Relative URIS**

In the context of network programming, resolving relative URIs is the process of converting a relative URI into an absolute URI that can be used to locate a resource on the internet. A relative URI is a URI that does not include a complete URL, but instead refers to a resource in relation to another URI. To resolve a relative URI, the following steps are typically followed:

1. Determine the base URI: The base URI is the URI to which the relative URI is relative. It can be the current document, or it can be specified explicitly.
2. Combine the base URI and the relative URI: This involves concatenating the paths of the two URIs, taking into account any special characters such as ".." (to move up a level in the directory structure) and "." (to refer to the current directory).
3. Normalize the resulting URI: This involves removing any unnecessary elements, such as redundant slashes and dot segments.
4. Encode the resulting URI: This involves converting any non-ASCII characters and special characters into their corresponding escape sequences.

*import java.net.URI;*

*import java.net.URISyntaxException;*

*public class ResolveURI {*

*public static void main(String[] args) {*

*try {*

*// Define the base URI*

*URI baseURI = new URI("http://www.example.com/path/to/document.html");*

*// Define the relative URI*

*URI relativeURI = new URI("../images/image.jpg");*

*// Resolve the relative URI against the base URI*

*URI absoluteURI = baseURI.resolve(relativeURI);*

*// Print the resulting absolute URI*

*System.out.println("Resolved URI: " + absoluteURI);*

*} catch (URISyntaxException e) {*

*System.out.println("Invalid URI: " + e.getMessage());*

*}*

*}*

*}*

**Utility Method in URI**

The java.net.URI class in Java provides several utility methods that can be used to work with URIs. Here are some of the most commonly used utility methods of the URI class:

1. parse(String uri) - This method parses the given URI string and returns a new URI object. If the given string is not a valid URI, a URISyntaxException is thrown.
2. resolve(URI uri) - This method resolves the given URI against the current URI and returns a new URI object that represents the resulting URI. This method can be used to resolve relative URIs against a base URI.
3. normalize() - This method normalizes the URI by removing dot segments ("." and "..") from the path component. This can be useful to ensure that two URIs that refer to the same resource are represented in the same way.
4. isAbsolute() - This method returns true if the URI is an absolute URI, and false if it is a relative URI.
5. getScheme() - This method returns the scheme component of the URI, such as http, ftp, file, etc.
6. getAuthority() - This method returns the authority component of the URI, which typically includes the username, password, host name, and port number.
7. getPath() - This method returns the path component of the URI, which represents the hierarchical path to the resource.
8. getQuery() - This method returns the query component of the URI, which is a string of parameters or data associated with the resource.
9. getFragment() - This method returns the fragment component of the URI, which represents a secondary resource or sub-component of the main resource.
10. toURL() - This method converts the URI to a java.net.URL object, which can be used to open a connection to the resource represented by the URI.

**X-WWW-FORM-URIENCODED**

x-www-form-urlencoded is a widely used method of encoding data in an HTTP POST request, particularly for sending form data. It involves formatting data as key-value pairs separated by an equal sign (=), with each pair separated by an ampersand (&) character. The keys and values are URL encoded, meaning any special characters are replaced with their corresponding percent-encoded values, and spaces are replaced with the plus sign (+). This encoding method is supported by most web servers and programming languages, and the server receiving the request can decode the data to retrieve the original key-value pairs.

1. x-www-form-urlencoded is a method used to encode data in an HTTP POST request.
2. It is one of the default encoding types used by HTML forms.
3. In this encoding method, data is sent as key-value pairs separated by an equal sign (=), and each pair is separated by an ampersand (&) character.
4. The keys and values are URL encoded, which means that any special characters in the data are replaced by their corresponding percent-encoded values.
5. Spaces are also encoded by replacing them with the plus sign (+).
6. For example, if you submit a form with two fields "name" and "email" with the values "John Doe" and "[john@example.com](mailto:john@example.com)" respectively, the encoded data in the request body would look like this:

name=John+Doe&email=john%40example.com

1. The server receiving the request can then decode the data to retrieve the original key-value pairs.
2. This encoding method is commonly used to send form data in HTTP requests.
3. It is supported by most web servers and programming languages.

**URL Encoder**

URL encoding is a technique used to convert characters in a URL or query string to a format that can be transmitted over the internet. It is used to ensure that special characters and non-ASCII characters are properly represented in a URL, which otherwise might cause errors or unexpected behavior. Here are some key points about URL encoding:

1. URL encoding is typically used when sending data in an HTTP request or when constructing a URL that contains special characters.
2. URL encoding replaces characters that are not allowed in a URL with a % symbol followed by two hexadecimal digits representing the ASCII code of the character.
3. Commonly encoded characters include spaces, commas, question marks, ampersands, and equals signs.
4. The encoding scheme used for URL encoding is typically UTF-8, which can represent the full range of Unicode characters.
5. In Java, the java.net.URLEncoder class provides a method called encode() that can be used to URL-encode a string.
6. When decoding a URL-encoded string, the java.net.URLDecoder class can be used, which has a method called decode().
7. It is important to always properly URL-encode special characters and non-ASCII characters in order to ensure the integrity of the URL and avoid errors or unexpected behavior.

*import java.net.URLEncoder;*

*public class URLEncodingExample {*

*public static void main(String[] args) throws Exception {*

*String originalString = "http://www.example.com/search?q=java tutorial";*

*String encodedString = URLEncoder.encode(originalString, "UTF-8");*

*System.out.println("Encoded string: " + encodedString);*

*}*

*}*

**URI DECODER**

URI Decoder is a utility class in Java that provides methods for decoding Uniform Resource Identifiers (URIs) that have been encoded using percent-encoding. URIs may contain characters that are not allowed or have special meanings in certain contexts, so URI Decoder is used to convert these encoded characters back to their original form. The decoding process involves replacing percent-encoded sequences (which begin with the '%' character followed by two hexadecimal digits) with their corresponding characters. URI Decoder can be used to decode the entire URI or just a specific component of the URI, such as the path or query string. It is commonly used in web applications to decode URLs received from the client before processing them.

A URI (Uniform Resource Identifier) decoder is a tool that can be used to decode a URI that has been encoded in a specific format. The purpose of URI decoding is to translate a string of characters that have been encoded using percent-encoding (also known as URL encoding) into their original form.

The key points to keep in mind when using a URI decoder are:

1. URI encoding uses percent-encoding, which means that certain characters are represented by a percent sign (%) followed by two hexadecimal digits. For example, the space character is encoded as %20.
2. A URI decoder is used to decode a URI that has been encoded in this way. Decoding involves replacing the percent-encoded characters with their original form.
3. A URI decoder can be used to decode the query string parameters in a URL, which are often encoded using percent-encoding. This can be useful for extracting data from a URL.
4. It is important to use a URI decoder that is compatible with the specific encoding format used. For example, UTF-8 is a common encoding format for URI components, and a decoder that supports UTF-8 should be used to decode URIs that have been encoded using this format.
5. URI decoding can be used in combination with URI encoding to create URLs that are both human-readable and machine-readable. By encoding certain characters and decoding them when necessary, URLs can be created that are both easy to read and process.
6. Top of Form

*import java.net.URLDecoder;*

*import java.nio.charset.StandardCharsets;*

*public class URIDecoderExample {*

*public static void main(String[] args) {*

*try {*

*String encodedURI = "https://example.com/search?q=hello%20world";*

*String decodedURI = URLDecoder.decode(encodedURI, StandardCharsets.UTF\_8);*

*System.out.println(decodedURI);*

*} catch (Exception e) {*

*e.printStackTrace();*

*}*

*}*

*}*

**Proxies**

A proxy is a server that acts as an intermediary between a client and other servers, enabling the client to access resources such as websites, files, and other data indirectly. When a client sends a request to a proxy server, the proxy server forwards the request to the target server on behalf of the client, and then sends the response back to the client. Proxies can be used for various purposes such as improving network performance, bypassing content restrictions, enhancing security, and providing anonymity for the client.

Proxies have various uses, some of which include:

1. Anonymity: Proxies can be used to hide a user's IP address and location, making them anonymous while browsing the internet.
2. Content filtering: Proxies can be used to bypass content restrictions imposed by ISPs or governments by allowing users to access blocked websites and content.
3. Load balancing: Proxies can be used to distribute network traffic across multiple servers, improving network performance and preventing server overload.
4. Security: Proxies can be used to provide an additional layer of security by filtering incoming traffic and preventing attacks such as DDoS.
5. Caching: Proxies can store frequently accessed content on local servers, reducing the amount of traffic sent to the origin server and improving performance.
6. Debugging: Proxies can be used to intercept and inspect network traffic for debugging purposes, allowing developers to diagnose and fix network issues.
7. Geolocation: Proxies can be used to access region-specific content by routing traffic through servers located in the target region.

**Proxy Class**

In computer programming, a proxy class is a class that acts as a placeholder for another class. The proxy class has the same interface as the original class, allowing clients to interact with it in the same way. When a client sends a request to the proxy class, the proxy class intercepts the request and forwards it to the original class, which performs the requested action. The proxy class can also add additional functionality such as caching, security, or logging. Proxy classes are commonly used in object-oriented programming to provide a layer of abstraction between clients and the underlying implementation, improving maintainability and flexibility. They can also be used for remote procedure calls, where the proxy class communicates with a remote server to execute methods on behalf of the client.

*import java.net.\*;*

*public class ProxyDemo {*

*public static void main(String[] args) throws Exception {*

*String urlString = "https://www.google.com";*

*URL url = new URL(urlString);*

*// create a proxy instance with the host and port number*

*Proxy proxy = new Proxy(Proxy.Type.HTTP, new InetSocketAddress("proxy.example.com", 8080));*

*// open the connection using the proxy*

*HttpURLConnection connection = (HttpURLConnection)url.openConnection(proxy);*

*// set the request method and other properties*

*connection.setRequestMethod("GET");*

*connection.setConnectTimeout(5000);*

*connection.setReadTimeout(5000);*

*// read the response from the server*

*int responseCode = connection.getResponseCode();*

*System.out.println("Response code: " + responseCode);*

*// close the connection*

*connection.disconnect();*

*}*

*}*

**Proxy Selector Class**

ProxySelector is a Java class that provides a way to configure and choose which proxy servers to use for connecting to a particular network resource. It maintains a list of available proxy servers and selects the most appropriate one based on the application's network configuration and proxy server preferences. It is particularly useful when an application needs to connect to a remote server through an intermediate proxy server, as is often the case in enterprise networks.

Here are some uses of the ProxySelector class:

1. Proxy configuration: The ProxySelector class is used to configure and select a proxy server for an application. It provides methods for adding and removing proxy servers, and for querying the current proxy server settings.
2. Network traffic monitoring: The ProxySelector class can be used to monitor network traffic and to log requests and responses. This is useful for debugging network-related issues in an application.
3. Privacy and security: The ProxySelector class can be used to enforce privacy and security policies in an application. For example, it can be used to route all network traffic through a secure proxy server to prevent unauthorized access to sensitive information.
4. Load balancing: The ProxySelector class can be used to implement load balancing in an application. It can be configured to distribute network traffic across multiple proxy servers to improve performance and scalability.
5. Network optimization: The ProxySelector class can be used to optimize network performance by selecting the most appropriate proxy server for a particular network connection. For example, it can be configured to use a proxy server that is geographically closer to the network resource being accessed to reduce latency and improve throughput.

*import java.net.\*;*

*public class ProxySelectorDemo {*

*public static void main(String[] args) throws Exception {*

*String urlString = "https://www.google.com";*

*URL url = new URL(urlString);*

*// configure the proxy selector to use a proxy server*

*ProxySelector.setDefault(new ProxySelector() {*

*@Override*

*public List<Proxy> select(URI uri) {*

*List<Proxy> proxies = new ArrayList<>();*

*proxies.add(new Proxy(Proxy.Type.HTTP, new InetSocketAddress("proxy.example.com", 8080)));*

*return proxies;*

*}*

*@Override*

*public void connectFailed(URI uri, SocketAddress sa, IOException ioe) {*

*System.out.println("Connection failed: " + ioe.getMessage());*

*}*

*});*

*// open a connection to the URL using the default proxy selector*

*HttpURLConnection connection = (HttpURLConnection)url.openConnection();*

*// set the request method and other properties*

*connection.setRequestMethod("GET");*

*connection.setConnectTimeout(5000);*

*connection.setReadTimeout(5000);*

*// read the response from the server*

*int responseCode = connection.getResponseCode();*

*System.out.println("Response code: " + responseCode);*

*// close the connection*

*connection.disconnect();*

*}*

*}*

**Communicating With Server-Side Program Through Get**

Communicating with a server-side program through the GET method involves constructing a URL that contains the necessary parameters to request a resource, sending a GET request to the server using the constructed URL, receiving a response from the server-side program containing the requested data, and parsing the response to extract the information needed. The GET method is used to retrieve resources from the server and is a widely-used HTTP method that can be implemented using a variety of tools and programming languages. By using GET, clients can easily retrieve information from server-side programs without needing to establish a persistent connection with the server.

**Benefits:**

* Easy to implement and widely supported by most programming languages and tools
* Caching can be used to improve performance by storing the response from the server-side program for future requests
* No need to establish a persistent connection with the server, reducing resource usage and improving scalability
* GET requests can be bookmarked and shared, making it easy to access resources repeatedly without having to remember the URL or parameters

**Uses**

* Retrieving data from a server-side database, such as a list of products or user information
* Accessing web services or APIs that provide information or functionality, such as weather data or payment processing
* Sending parameters to a server-side script to perform an action, such as adding an item to a shopping cart or performing a search
* Debugging and troubleshooting server-side issues by examining the response data returned from the server

**Accessing Password Protected Sites**

Accessing password protected sites requires providing valid authentication credentials, such as a username and password, to the server. Typically, when a user enters their credentials, they are sent to the server over a secure connection, and the server checks the credentials against its database. If the credentials are valid, the server sends a cookie or session token back to the user's browser, which is stored and sent with subsequent requests to the server. This allows the user to access protected pages or resources on the site without having to enter their credentials again. To access password protected sites programmatically, the authentication process can be automated using tools such as Selenium or cURL, which can simulate user interactions and send authentication requests to the server. It's important to note that accessing password protected sites without authorization is illegal and can result in legal consequences.

**The Authenticator Class**

The Authenticator class in Java is a useful tool for implementing authentication in network programs. When a network program makes a request to a server that requires authentication, the Authenticator class can be used to provide the necessary credentials automatically. The Authenticator class has a single method, getPasswordAuthentication(), that takes a hostname, port, authentication scheme, and prompt message as arguments. When a request is made to a server that requires authentication, the getPasswordAuthentication() method is called and returns a PasswordAuthentication object containing the username and password for the requested server. These credentials can be stored and used in subsequent requests to the server, allowing the network program to access protected resources without requiring the user to enter their credentials repeatedly. Using the Authenticator class can simplify the authentication process in network programs and improve the user experience.

**Uses of Authenticator Class**

1. Providing authentication credentials for network requests: The Authenticator class can be used to provide authentication credentials for network requests that require them. This can simplify the authentication process and improve the user experience.
2. Implementing custom authentication schemes: The Authenticator class can be extended to implement custom authentication schemes, such as token-based authentication or OAuth.
3. Securing access to sensitive information: The Authenticator class can be used to secure access to sensitive information, such as passwords or private keys, by requiring authentication before granting access.
4. Enforcing access controls: The Authenticator class can be used to enforce access controls by requiring different levels of authentication or authorization for different users or roles.
5. Supporting multi-factor authentication: The Authenticator class can be used to support multi-factor authentication, where users are required to provide multiple types of credentials, such as a password and a security token or biometric data, to access a system or service.

*//Example to access password protected site*

*import java.net.Authenticator;*

*import java.net.PasswordAuthentication;*

*import java.net.URL;*

*import java.io.BufferedReader;*

*import java.io.InputStreamReader;*

*public class PasswordProtectedSiteAccess {*

*public static void main(String[] args) throws Exception {*

*// Set the authentication credentials for the site*

*String username = "myusername";*

*String password = "mypassword";*

*String siteURL = "https://www.example.com";*

*// Set the Authenticator for the site*

*Authenticator.setDefault(new Authenticator() {*

*protected PasswordAuthentication getPasswordAuthentication() {*

*return new PasswordAuthentication(username, password.toCharArray());*

*}*

*});*

*// Create a URL object for the protected site*

*URL url = new URL(siteURL);*

*// Open a connection to the site and read the content*

*BufferedReader in = new BufferedReader(new InputStreamReader(url.openStream()));*

*String inputLine;*

*while ((inputLine = in.readLine()) != null) {*

*System.out.println(inputLine);*

*}*

*in.close();*

*}*

*}*

**The Jpassword Field Class**

JPasswordField is a class in the Java Swing framework that represents a text component used for entering passwords. It is a specialized extension of the JTextField class, designed to handle sensitive information such as passwords or other confidential data. When the user types into a JPasswordField, the text is not displayed on the screen, but rather replaced with an asterisk or other character to prevent onlookers from seeing the entered text. The text is stored as an array of characters rather than a String, providing an additional layer of security. JPasswordField also supports features such as customizable echo characters, enabling the developer to configure the appearance of the text while it is being entered. Overall, JPasswordField provides a secure and user-friendly way to handle sensitive information in Java Swing applications.

*//Example to download password protected web page*

*import java.io.\*;*

*import java.net.\*;*

*import java.util.Base64;*

*public class PasswordProtectedWebPageDownloader {*

*public static void main(String[] args) {*

*try {*

*// Set the URL of the password-protected web page*

*String url = "https://example.com/protected-page";*

*// Set the username and password for HTTP basic authentication*

*String username = "myusername";*

*String password = "mypassword";*

*// Encode the username and password in base64*

*String authString = username + ":" + password;*

*String encodedAuthString = Base64.getEncoder().encodeToString(authString.getBytes());*

*// Create an HTTP connection to the web page URL*

*HttpURLConnection connection = (HttpURLConnection) new URL(url).openConnection();*

*// Set the HTTP authentication header*

*connection.setRequestProperty("Authorization", "Basic " + encodedAuthString);*

*// Read the response from the server*

*BufferedReader reader = new BufferedReader(new InputStreamReader(connection.getInputStream()));*

*String line;*

*StringBuilder responseBuilder = new StringBuilder();*

*while ((line = reader.readLine()) != null) {*

*responseBuilder.append(line);*

*}*

*reader.close();*

*String response = responseBuilder.toString();*

*// Print the response to the console*

*System.out.println(response);*

*// Disconnect the HTTP connection*

*connection.disconnect();*

*} catch (IOException e) {*

*e.printStackTrace();*

*}*

*}*

*}*

Bottom of Form