Unit 2: Internet Addresses

The NetworkInterface Class:

The NetworkInterface class in Java represents a local IP address, which can either be tied to a physical network interface, such as an Ethernet card, or a virtual interface associated with the machine's other IP addresses.

This class is integral for network programming as it provides methods to enumerate all local addresses and create InetAddress objects from them. These InetAddress objects are subsequently used for creating sockets, server sockets, and other network-related entities.

Key Features and Methods:

1) Representation of Network Interfaces:

- The NetworkInterface class encapsulates both physical interfaces (e.g., Ethernet cards) and virtual interfaces bound to the same physical hardware.

2) Factory Methods:

- These methods provide NetworkInterface objects without needing to construct them directly. This is similar to the InetAddress class.
- **getByName(String name):** This method returns a NetworkInterface object corresponding to the specified name. If no interface matches the given name, it returns null and throws a SocketException if an error occurs during the search.

```
NetworkInterface ni = NetworkInterface.getByName("eth0");
if (ni == null) {
    System.err.println("No such interface: eth0");
}
```

```
import java.net.NetworkInterface;
import java.net.SocketException;

public class CheckNetworkInterface {
    public static void main(String[] args) {
        try {
            NetworkInterface ni = NetworkInterface.getByName("eth0");

            if (ni == null) {
                System.err.println("No such interface: eth0");
            } else {
                System.out.println("Interface found: " + ni);
            }
        } catch (SocketException e) {
            e.printStackTrace();
        }
    }
}
```

- **getByInetAddress(InetAddress address):** This method returns the NetworkInterface object for a given IP address. If no network interface is bound to the IP address, it returns null and may throw a SocketException.

  ```
  InetAddress local = InetAddress.getByName("127.0.0.1");
  NetworkInterface ni = NetworkInterface.getByInetAddress(local);
  if (ni == null) {
      System.err.println("No local loopback address.");
  }
  ```

- **getNetworkInterfaces():** This method provides an enumeration of all network interfaces on the local host.

  ```
  Enumeration<NetworkInterface> interfaces =
  NetworkInterface.getNetworkInterfaces();
  while (interfaces.hasMoreElements()) {
      NetworkInterface ni = interfaces.nextElement();
      System.out.println(ni);
  }
  ```

3) Getter Methods:

- **getInetAddresses():** Returns an enumeration of all InetAddress objects bound to the interface.
- **getName():** Returns the name of the network interface (e.g., "eth0" or "lo").
- **getDisplayName():** Provides a more user-friendly name for the interface. On Unix systems, this often matches the name, while on Windows, it might be a more descriptive name like "Local Area Connection".

Example: Program that lists all network interfaces on the local host:

```
import java.net.*;

import java.util.*;

public class InterfaceLister {

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

      Enumeration<NetworkInterface> interfaces =
NetworkInterface.getNetworkInterfaces();

      while (interfaces.hasMoreElements()) {

         NetworkInterface ni = interfaces.nextElement();

         System.out.println(ni);

      }

   }

}
```

Some Useful Programs:

1) SpamCheck:

- A number of services monitor spammers, and inform clients whether a host attempting to connect to them is a known spammer or not.
- These real-time blackhole lists need to respond to queries extremely quickly, and process a very high load.
- The nature of the problem requires that the response be fast, and ideally it should be cacheable. Furthermore, the load should be distributed across many servers, ideally ones located around the world.

2) Processing Web Server Logfiles:

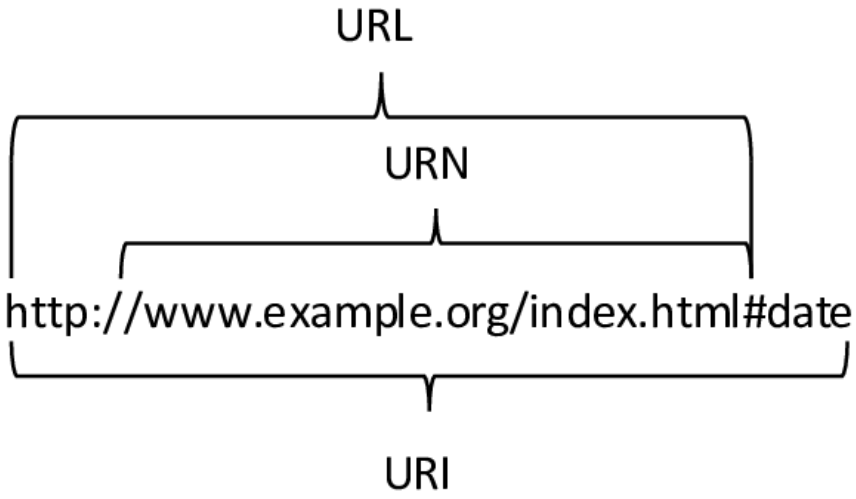
- Web server logs track the hosts that access a website. By default, the log reports the IP addresses of the sites that connect to the server.
- Most web servers have an option to store hostnames instead of IP addresses, but this can hurt performance because the server needs to make a DNS request for each hit.
- It is much more efficient to log the IP addresses and convert them to hostnames at a later time, when the server isn't busy or even on another machine completely.

Unit 3: URLs & URIs

URLs:

A Uniform Resource Identifier (URI) is a string that identifies a resource. This resource could be a file, email address, news message, book, person's name, internet host, stock price, or more.

A single resource may have various representations (e.g., plain text, XML, PDF) and may be available in different languages.



URL:

- A URL is a type of URI that provides a specific network location for a resource.
- A URI identifies a resource but does not necessarily provide its location or access method.
- In the physical world, a URI is like a book title, while a URL is like a library location.
- In Java, java.net.URI identifies resources, and java.net.URL can identify and retrieve resources.
- A URL includes the protocol, hostname/IP address, and path to the resource.
- The protocol part of a URL specifies the access method (e.g., HTTP, FTP).

- The host part specifies the server name or IP address.

Relative URL:

- Relative URLs are a way to link to web resources using paths relative to the current document's location.
- They are especially useful for maintaining links within a website, as they allow for easier site maintenance and portability.
- If the relative link begins with a /, then it is relative to the document root instead of relative to the current file.

Example: <a href="/projects/ipv6/">

Why Use Relative URLs?(Advantages)

1. **Portability:** Easier to move or copy entire directories without needing to update URLs.
2. **Maintenance:** Simplifies the process of updating URLs when directories are moved or renamed.
3. **Development:** Simplifies local development and testing since the URLs do not depend on the specific domain or directory structure.

The java.net.URL class is an abstraction of a Uniform Resource Locator such as http://www.lolcats.com/ or ftp://ftp.redhat.com/pub/. It extends java.lang.Object, and it is a final class that cannot be subclassed. Rather than relying on inheritance to configure instances for different kinds of URLs, it uses the strategy design pattern.

The URL Class:

- java.net.URL is an abstraction of a Uniform Resource Locator (URL).
- Extends java.lang.Object.
- It is a final class and cannot be subclassed.
- URLs are represented as objects with fields including: (protocol, Hostname, Port, Path, Query string, Fragment identifier)

Example of how to use the URL class in a Java program:

```
import java.net.MalformedURLException;

import java.net.URL;

public class URLExample {

    public static void main(String[] args) {

        try {

            // Constructing a URL object

            URL url = new
URL("http://www.example.com:80/docs/resource1.html?name=networking#DOWNLOA
DING");



            // Accessing different parts of the URL

            System.out.println("URL: " + url);

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

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

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

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

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

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



        } catch (MalformedURLException e) {

            e.printStackTrace();

        }
```

```
    }
}
```

Constructing a URL from a string:

Instances of java.net.URL can be constructed using various constructors, depending on the available information:

- Single String Constructor.
- Protocol, Hostname, and File Constructor.
- Protocol, Host, Port, and File Constructor.
- Base URL and Relative URL Constructor.

example(Protocol, Host, Port, and File Constructor):

```
import java.net.MalformedURLException;

import java.net.URL;

public class URLExample {

    public static void main(String[] args) {

        try {

            URL url = new URL("http", "www.example.com", 8080,
"/path/to/resource.html?query=123#section2");



            System.out.println("URL: " + url);

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

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

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

            System.out.println("File: " + url.getFile());
```

```
            System.out.println("Path: " + url.getPath());
            System.out.println("Query: " + url.getQuery());
            System.out.println("Fragment: " + url.getRef());

        } catch (MalformedURLException ex) {
            System.err.println("Malformed URL: " + ex.getMessage());
        }
    }
}
```

Which protocols does a virtual machine support?

```
import java.net.*;
public class ProtocolTester {
 public static void main(String[] args) {
 testProtocol("http://www.adc.org");
 testProtocol("https://www.amazon.com/exec/obidos/order2/");
 testProtocol("ftp://ibiblio.org/pub/languages/java/javafaq/");
 testProtocol("mailto:elharo@ibiblio.org");
 testProtocol("telnet://dibner.poly.edu/");
 testProtocol("file:///etc/passwd");
 testProtocol("gopher://gopher.anc.org.za/");
testProtocol("ldap://ldap.itd.umich.edu/o=University%20of%20Michigan,c=US?postalAd
dress");
testProtocol("jar:http://cafeaulait.org/books/javaio/ioexamples/javaio.jar!"+"/com/macfaq/
io/StreamCopier.class");
```

```
testProtocol("nfs://utopia.poly.edu/usr/tmp/");
testProtocol("jdbc:mysql://luna.ibiblio.org:3306/NEWS");
testProtocol("rmi://ibiblio.org/RenderEngine");
testProtocol("doc:/UsersGuide/release.html");
testProtocol("netdoc:/UsersGuide/release.html");
testProtocol("systemresource://www.adc.org/+/index.html");
testProtocol("verbatim:http://www.adc.org/");
}
private static void testProtocol(String url) {
try {
URL u = new URL(url);
System.out.println(u.getProtocol() + " is supported");
} catch (MalformedURLException ex) {
String protocol = url.substring(0, url.indexOf(':'));
System.out.println(protocol + " is not supported");
}
 }
}
```

Retrieving Data from a URL:

The URL class has several methods that retrieve data from a URL:

```
public InputStream openStream() throws IOException
public URLConnection openConnection() throws IOException
public URLConnection openConnection(Proxy proxy) throws IOException
public Object getContent() throws IOException
public Object getContent(Class[] classes) throws IOException
```

- The most basic and most commonly used of these methods is openStream(), which returns an InputStream from which you can read the data.
- If you need more control over the download process, call openConnection() instead, which gives you a URLConnection which you can configure, and then get an InputStream from it.

Example: reading a url data through,

`public InputStream openStream() throws IOException` method.

```
import java.io.IOException;
import java.io.InputStream;
import java.net.URL;

public class URLContentReader {
   public static void main(String[] args) {
      try {
         URL u = new URL("https://example.com/");

         InputStream in = u.openStream();

         int c;

         while ((c = in.read()) != -1) {
            System.out.write(c);
         }

         in.close();
      } catch (IOException ex) {
         System.err.println(ex);
      }
   }
}
```

- If you need more control over the download process, call openConnection() instead, which gives you a URLConnection which you can configure, and then get an InputStream from it.

- you can ask the URL for its content with getContent() which may give you a more complete object such as String or an Image. Then again, it may just give you an InputStream anyway.

Example of `public URLConnection openConnection() throws IOException` method,

```
import java.io.IOException;
import java.io.InputStream;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLConnection;

public class URLConnectionExample {
    public static void main(String[] args) {
        try {
            URL url = new URL("https://news.ycombinator.com/");

            try {
                URLConnection urlConnection = url.openConnection();

                InputStream inputStream = urlConnection.getInputStream();

                int byteRead;

                while ((byteRead = inputStream.read()) != -1) {
                    System.out.write(byteRead);
                }

                inputStream.close();

            } catch (IOException ex) {
                System.err.println("IOException: " + ex.getMessage());
            }

        } catch (MalformedURLException ex) {
            System.err.println("MalformedURLException: " + ex.getMessage());
        }
    }
}
```

Splitting a URL into Pieces:

URLs are composed of five pieces:

1. The scheme, also known as the protocol
2. The authority
3. The path
4. The fragment identifier, also known as the section or ref
5. The query string

Example: The parts of a URL

```
import java.net.*;

public class URLSplitter {

 public static void main(String args[]) {

 for (int i = 0; i < args.length; i++) {

 try {

 URL u = new URL(args[i]);

 System.out.println("The URL is " + u);

 System.out.println("The scheme is " + u.getProtocol());

 System.out.println("The user info is " + u.getUserInfo());

 String host = u.getHost();

 if (host != null) {

 int atSign = host.indexOf('@');

 if (atSign != -1) host = host.substring(atSign+1);

 System.out.println("The host is " + host);

 } else {

 System.out.println("The host is null.");

 }

 System.out.println("The port is " + u.getPort());
```

```
 System.out.println("The path is " + u.getPath());
 System.out.println("The ref is " + u.getRef());
 System.out.println("The query string is " + u.getQuery());
 } catch (MalformedURLException ex) {
 System.err.println(args[i] + " is not a URL I understand.");
 }
 System.out.println();
}
}
}
```

Equality and Comparison:

The URL class contains the usual equals() and hashCode() methods. These behave almost as you'd expect. Two URLs are considered equal if and only if both URLs point to the same resource on the same host, port, and path, with the same fragment identifier and query string.

Example:

Are http://www.ibiblio.org and http://ibiblio.org the same?

```
import java.net.*;
public class URLEquality {
 public static void main (String[] args) {
 try {
 URL www = new URL ("http://www.ibiblio.org/");
 URL ibiblio = new URL("http://ibiblio.org/");
```

```
 if (ibiblio.equals(www)) {
 System.out.println(ibiblio + " is the same as " + www);
 } else {
 System.out.println(ibiblio + " is not the same as " + www);
 }
 } catch (MalformedURLException ex) {
 System.err.println(ex);
 }
 }
}
```

Conversion:

URL has three methods that convert an instance to another form:

- toString()
- toExternalForm()
- toURI()

The URI Class:

- A URI is a generalization of a URL that includes not only Uniform Resource Locators but also Uniform Resource Names (URNs).
- Most URIs used in practice are URLs, but most specifications and standards such as XML are defined in terms of URIs.
- In Java, URIs are represented by the java.net.URI class. This class differs from the java.net.URL class in three important ways:
- The URI class is purely about identification of resources and parsing of URIs. It provides no methods to retrieve a representation of the resource identified by its URI.

- The URI class is more conformant to the relevant specifications than the URL class.
- A URI object can represent a relative URI. The URL class absolutizes all URIs before storing them.

Constructing a URI:

URIs are built from strings. You can either pass the entire URI to the constructor in a single string, or the individual pieces:

```
public URI(String uri) throws URISyntaxException
public URI(String scheme, String schemeSpecificPart, String fragment)
    throws URISyntaxException
public URI(String scheme, String host, String path, String fragment)
    throws URISyntaxException
public URI(String scheme, String authority, String path, String query,
    String fragment) throws URISyntaxException
public URI(String scheme, String userInfo, String host, int port,
    String path, String query, String fragment) throws URISyntaxException
```

Unlike the URL class, the URI class does not depend on an underlying protocol handler. As long as the URI is syntactically correct, Java does not need to understand its protocol in order to create a representative URI object. Thus, unlike the URL class, the URI class can be used for new and experimental URI schemes.

The first constructor creates a new URI object from any convenient string.

For example:

```
URI voice = new URI("tel:+1-800-9988-9938");
URI web   = new URI("http://www.xml.com/pub/a/2003/09/17/stax.html#id=_hbc");
URI book  = new URI("urn:isbn:1-565-92870-9");
```

The Parts of the URI:

A URI reference has up to three parts: a scheme, a scheme-specific part, and a fragment identifier. The general format is:

scheme:scheme-specific-part:fragment

If the scheme is omitted, the URI reference is relative. If the fragment identifier is omitted, the URI reference is a pure URI. The URI class has getter methods that return these three parts of each URI object.

```
public String getScheme()
public String getSchemeSpecificPart()
public String getRawSchemeSpecificPart()
public String getFragment()
public String getRawFragment()
```

These methods all return null if the particular URI object does not have the relevant component: for example, a relative URI without a scheme or an http URI without a fragment identifier.

- A URI that has a scheme is an absolute URI.
- A URI without a scheme is relative.
- The isAbsolute() method returns true if the URI is absolute, false if it's relative:

Resolving Relative URIs:

The URI class has three methods for converting back and forth between relative and absolute URIs:

```
public URI resolve(URI uri)
public URI resolve(String uri)
public URI relativize(URI uri)
```

The resolve() methods compare the uri argument to this URI and use it to construct a new URI object that wraps an absolute URI. For example, consider these three lines of code:

```
URI absolute = new URI("http://www.example.com/");
URI relative = new URI("images/logo.png");
URI resolved = absolute.resolve(relative);
```

After they've executed, resolved contains the absolute URI http://www.example.com/images/logo.png

String Representations:

Two methods convert URI objects to strings, toString() and toASCIIString():

```
public String toString()
public String toASCIIString()
```

The toString() method returns an unencoded string form of the URI (i.e., characters like é and \ are not percent escaped). Therefore, the result of calling this method is not guaranteed to be a syntactically correct URI, though it is in fact a syntactically correct IRI. This form is sometimes useful for display to human beings, but usually not for retrieval.

The toASCIIString() method returns an encoded string form of the URI. Characters like é and \ are always percent escaped whether or not they were originally escaped. This is the string form of the URI you should use most of the time. Even if the form returned by toString() is more legible for humans, they may still copy and paste it into areas that are not expecting an illegal URI. toASCIIString() always returns a syntactically correct URI.

x-www-form-urlencoded:

One of the challenges faced by the designers of the Web was dealing with the differences between operating systems. These differences can cause problems with URLs: for ex- ample, some operating systems allow spaces in filenames; some don't. Most operating systems won't complain about a # sign in a filename; but in a URL, a # sign indicates that the filename has ended, and a fragment identifier follows.

Example: x-www-form-urlencoded strings(URL encoder).

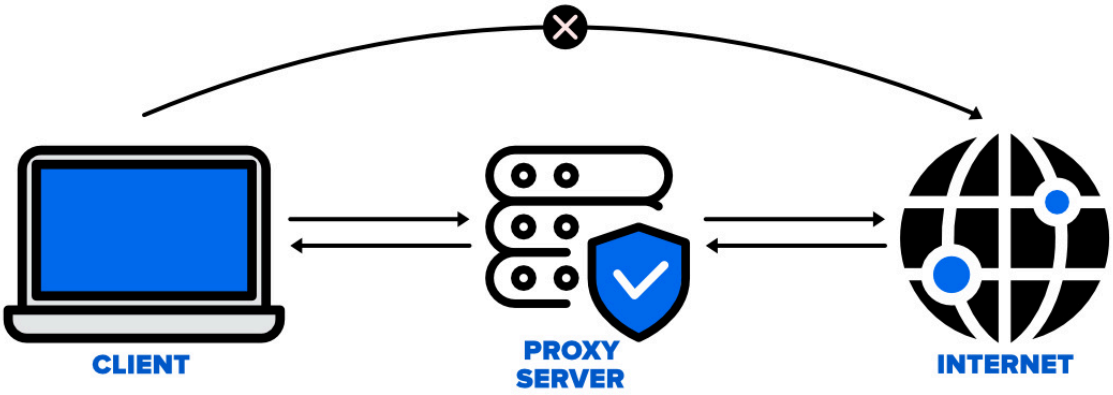
```
import java.io.*;

import java.net.*;

public class EncoderTest {

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

```
try {
System.out.println(URLEncoder.encode("This string has spaces",
"UTF-8"));
System.out.println(URLEncoder.encode("This*string*has*asterisks",
"UTF-8"));
System.out.println(URLEncoder.encode("This%string%has%percent%signs",
"UTF-8"));
System.out.println(URLEncoder.encode("This+string+has+pluses",
"UTF-8"));
System.out.println(URLEncoder.encode("This/string/has/slashes",
"UTF-8"));
System.out.println(URLEncoder.encode("This\"string\"has\"quote\"marks",
"UTF-8"));
System.out.println(URLEncoder.encode("This:string:has:colons",
"UTF-8"));
System.out.println(URLEncoder.encode("This~string~has~tildes",
"UTF-8"));
System.out.println(URLEncoder.encode("This(string)has(parentheses)",
"UTF-8"));
System.out.println(URLEncoder.encode("This.string.has.periods",
"UTF-8"));
System.out.println(URLEncoder.encode("This=string=has=equals=signs",
"UTF-8"));
System.out.println(URLEncoder.encode("This&string&has&ampersands",
"UTF-8"));
```

```
System.out.println(URLEncoder.encode("Thiséstringéhasé
non-ASCII characters", "UTF-8"));
} catch (UnsupportedEncodingException ex) {
throw new RuntimeException("Broken VM does not support UTF-8");
}
}
}
```

Proxies:

A proxy server acts as an intermediary between a client (such as a web browser or any application making network requests) and the destination server (often on the internet). Java programs using classes like URL can easily work through proxy servers. The Proxy class and URLConnection class in Java provide mechanisms to handle various proxy configurations, allowing Java applications to interact seamlessly with both HTTP and non-HTTP services via proxies.



System Properties:

For basic operations, all you have to do is set a few system properties to point to the addresses of your local proxy servers. If you are using a pure HTTP proxy, set http.proxyHost to the domain name or the IP address of your proxy server and http.proxyPort to the port of the proxy server (the default is 80).

Example:

```
System.setProperty("http.proxyHost", "192.168.254.254");
System.setProperty("http.proxyPort", "9000");
System.setProperty("http.nonProxyHosts", "java.oreilly.com|xml.oreilly.com");
```

If you are using an FTP proxy server, set the ftp.proxyHost, ftp.proxyPort, and ftp.nonProxyHosts properties in the same way.

The Proxy Class:

The Proxy class allows more fine-grained control of proxy servers from within a Java program. Specifically, it allows you to choose different proxy servers for different remote hosts. The proxies themselves are represented by instances of the java.net.Proxy class.

There are still only three kinds of proxies, HTTP, SOCKS, and direct connections (no proxy at all), represented by three constants in the Proxy.Type enum:

- Proxy.Type.DIRECT
- Proxy.Type.HTTP
- Proxy.Type.SOCKS

The ProxySelector Class:

Each running virtual machine has a single java.net.ProxySelector object it uses to locate the proxy server for different connections. The default ProxySelector merely inspects the various system properties and the URL's protocol to decide how to connect to different hosts. However, you can install your own subclass of ProxySelector in place of the default selector and use it to choose different proxies based on protocol, host, path, time of day, or other criteria.

The key to this class is the abstract select() method:

- public abstract List select(URI uri)

Example: A ProxySelector that remembers what it can connect to

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

public class LocalProxySelector extends ProxySelector {
   private List<URI> failed = new ArrayList<URI>();

   public static void main(String[] args) {
      LocalProxySelector proxySelector = new LocalProxySelector();

      URI uri = URI.create("http://example.com");

      List<Proxy> proxies = proxySelector.select(uri);

      for (Proxy proxy : proxies) {
         System.out.println("Proxy Type: " + proxy.type());
         SocketAddress address = proxy.address();
         if (address instanceof InetSocketAddress) {
            InetSocketAddress inetAddress = (InetSocketAddress) address;
            System.out.println("Proxy Hostname: " + inetAddress.getHostName());
            System.out.println("Proxy Port: " + inetAddress.getPort());
         }
      }
   }

   @Override
   public List<Proxy> select(URI uri) {
      List<Proxy> result = new ArrayList<Proxy>();

      if (failed.contains(uri) || !"http".equalsIgnoreCase(uri.getScheme())) {
         result.add(Proxy.NO_PROXY);
      } else {
         SocketAddress proxyAddress = new InetSocketAddress("proxy.example.com",
8000);
         Proxy proxy = new Proxy(Proxy.Type.HTTP, proxyAddress);
         result.add(proxy);
      }
```

```
        return result;
    }

    @Override
    public void connectFailed(URI uri, SocketAddress address, IOException ex) {
        failed.add(uri);
    }
}
```

Communicating with Server-Side Programs Through GET

The URL class makes it easy for Java applets and applications to communicate with server side programs such as CGIs, servlets, PHP pages, and others that use the GET method.

How GET Requests Work

Client Initiation: A client (e.g., a web browser) initiates a GET request by typing a URL into the address bar or by clicking on a hyperlink.

URL Structure: The URL includes the path to the server-side program and may contain query parameters that are appended to the URL. Query parameters are added after a question mark (?) and are typically in the form of key-value pairs, separated by ampersands (&).

Example:

http://example.com/script.php?name=John&age=30

Server Processing: The server receives the GET request and processes the query parameters. The server-side script (e.g., PHP, Python, Node.js) retrieves the parameters and performs the necessary actions (e.g., querying a database, processing data).

Response: The server sends back a response to the client, which could be an HTML page, JSON data, or other formats.

Accessing Password-Protected Sites:

Accessing password-protected sites in Java can be achieved using different authentication methods. This guide focuses on HTTP authentication, which Java natively supports, and touches on cookie-based authentication, which is more complex and site-specific.

1) The Authenticator Class:

The Authenticator class in the java.net package provides a mechanism to supply username and password credentials when accessing HTTP authentication-protected resources.

```
public abstract class Authenticator extends Object
```

Since Authenticator is abstract, it must be subclassed to provide specific implementations for obtaining credentials. The Authenticator class uses the static method requestPasswordAuthentication() to fetch credentials when needed:

```
public static PasswordAuthentication requestPasswordAuthentication(
    InetAddress address, int port, String protocol, String promp,
        String scheme) throws SecurityException
```

Parameters:

- InetAddress address: The host requires authentication.
- int port: The port number.
- String protocol: The application layer protocol (e.g., HTTP).
- String prompt: The server-provided prompt, often the realm name.
- String scheme: The authentication scheme (typically "basic").

To get more details about the request, the following methods can be used:

- protected final InetAddress getRequestingSite()
- protected final int getRequestingPort()
- protected final String getRequestingProtocol()
- protected final String getRequestingPrompt()
- protected final String getRequestingScheme()
- protected final String getRequestingHost()
- protected final String getRequestingURL()
- protected Authenticator.RequestorType getRequestorType()

The PasswordAuthentication Class:

PasswordAuthentication encapsulates the username and password used for authentication.

```
public final class PasswordAuthentication extends Object
```

The class is initialized with the username and password:

```
public PasswordAuthentication(String userName, char[] password)
```

Properties:

- username: A String representing the username.
- password: A char[] holding the password, allowing it to be erased after use to enhance security.

Methods:

- public String getUserName(): Returns the username.
- public char[] getPassword(): Returns the password.

The JPasswordField Class:

JPasswordField is a Swing component for secure password input, displaying entered characters as asterisks or bullets to obscure them.

```
public class JPasswordField extends JTextField
```

Features:

- Echo Character: Characters typed are not displayed as entered but as a generic echo character (e.g., '*').
- Password Storage: Stores passwords as char[] to allow secure deletion from memory.

Methods:

- public char[] getPassword(): Retrieves the password as a char[].

Unit 4: HTTP

- HTTP (Hypertext Transfer Protocol) is a standard protocol for web client-server communication and data transfer.
- Facilitates communication between a web client (e.g., browser) and a server.
- Transfers various data types, not limited to HTML.
- HTTP can handle different types of files and data formats. Examples include TIFF pictures, Microsoft Word documents, Windows .exe files, and any data that can be represented in bytes.
- Typing a URL like http://www.google.com and pressing Return involves complex interactions between the client and server.

The Protocol:

HTTP is the standard protocol for communication between web browsers and web servers. HTTP specifies how a client and server establish a connection, how the client requests data from the server, how the server responds to that request, and finally, how the connection is closed.

HTTP connections use the TCP/IP protocol for data trans- fer. For each request from client to server, there is a sequence of four steps:

1. The client opens a TCP connection to the server on port 80, by default; other ports may be specified in the URL.
2. The client sends a message to the server requesting the resource at a specified path. The request includes a header, and optionally (depending on the nature of the request) a blank line followed by data for the request.
3. The server sends a response to the client. The response begins with a response code, followed by a header full of metadata, a blank line, and the requested document or an error message.
4. The server closed the connection.

Each request and response has the same basic form: a header line, an HTTP header containing metadata, a blank line, and then a message body. A typical client request looks something like this:

```
GET /index.html HTTP/1.1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.8; rv:20.0)
 Gecko/20100101 Firefox/20.0
Host: en.wikipedia.org
Connection: keep-alive
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
```

Keep-Alive:

HTTP 1.0 opens a new connection for each request. In practice, the time taken to open and close all the connections in a typical web session can outweigh the time taken to transmit the data, especially for sessions with many small documents. This is even more problematic for encrypted HTTPS connections using SSL or TLS, because the hand- shake to set up a secure socket is substantially more work than setting up a regular socket.

In HTTP 1.1 and later, the server doesn't have to close the socket after it sends its response. It can leave it open and wait for a new request from the client on the same socket. Multiple requests and responses can be sent in series over a single TCP connection. However, the lockstep pattern of a client request followed by a server response remains the same.

Connection: Keep-Alive

You can control Java's use of HTTP Keep-Alive with several system properties:

- Set *http.keepAlive* to "true or false" to enable/disable HTTP Keep-Alive. (It is enabled by default.)
- Set *http.maxConnections* to the number of sockets you're willing to hold open at one time. The default is 5.
- Set *http.keepAlive.remainingData* to true to let Java clean up after abandoned connections (Java 6 or later). It is false by default.

- Set *sun.net.http.errorstream.enableBuffering* to true to attempt to buffer the relatively short error streams from 400- and 500-level responses, so the connection can be freed up for reuse sooner. It is false by default.
- Set *sun.net.http.errorstream.bufferSize* to the number of bytes to use for buffering error streams. The default is 4,096 bytes.
- Set *sun.net.http.error stream.timeout* to the number of milliseconds before timing out a read from the error stream. It is 300 milliseconds by default.

HTTP Methods:

HTTP methods define the actions that can be performed on resources in a web server. Each method is represented by a verb and affects how clients interact with resources. Here's an overview of the primary HTTP methods:

1. GET

- **Purpose:** Retrieve a representation of a resource.
- **Characteristics:**
  - **Safe:** GET requests are meant to retrieve data without side effects.
  - **Idempotent:** Repeating a GET request yields the same result.
  - **Cacheable:** Responses to GET requests can be cached.
  - **Bookmarkable:** URLs from GET requests can be saved and shared.
- **Usage:** Used for retrieving resources like web pages or API data.

2. POST

- **Purpose:** Submit data to the server for processing.
- **Characteristics:**
  - **Unsafe:** POST requests are used for operations that modify the state of the server.
  - **Non-idempotent:** Repeating a POST request may result in different outcomes.
  - **Not Cacheable:** Responses to POST requests are generally not cached.
  - **Not Bookmarkable:** URLs from POST requests cannot be saved for later use.
- **Usage:** Used for actions such as creating new resources or submitting form data.

3. PUT

- **Purpose:** Upload a representation of a resource to the server.
- **Characteristics:**
    - **Idempotent:** Repeating a PUT request has the same effect as making it once.
    - **Not Safe:** PUT requests modify the state of the server.
    - **Not Cacheable:** Responses to PUT requests are generally not cached.
    - **Not Bookmarkable:** URLs from PUT requests are not typically saved.
- **Usage:** Used for updating or creating resources at a specified URL.

4. DELETE

- **Purpose:** Remove a resource from the server.
- **Characteristics:**
    - **Idempotent:** Repeating a DELETE request has the same effect as making it once.
    - **Not Safe:** DELETE requests alter the state of the server.
    - **Not Cacheable:** Responses to DELETE requests are generally not cached.
    - **Not Bookmarkable:** URLs from DELETE requests are not typically saved.
- **Usage:** Used for deleting resources from the server.

The Request Body:

The request body in HTTP methods like POST and PUT is a crucial part of how data is transmitted between a client and server. It contains the actual data being sent as part of the request. This section will break down how the request body works, how to structure it, and what it contains.

the Request Body is Structured:

When sending data in the request body, the format and content type are defined by the HTTP headers. Here's how the request body is structured for different HTTP methods and content types:

1. POST Request: POST requests are used to submit data to be processed by the server. The request body often contains form data or other types of data that the server will process.

Example POST Request:

```
POST /cgi-bin/register.pl HTTP/1.0
Date: Sun, 27 Apr 2013 12:32:36
Host: www.cafeaulait.org
Content-Type: application/x-www-form-urlencoded
Content-Length: 54

username=Elliotte+Harold&email=elharo%40ibiblio.org
```

2. PUT Request: PUT requests are used to update or create a resource at a specified URL. The request body contains the new representation of the resource.

Example PUT Request:

```
PUT /blog/software-development/the-power-of-pomodoros/ HTTP/1.1
Host: elharo.com
User-Agent: AtomMaker/1.0
Authorization: Basic ZGFmZnk6c2VjZXJldA==
Content-Type: application/atom+xml;type=entry
Content-Length: 322

<?xml version="1.0"?>
<entry xmlns="http://www.w3.org/2005/Atom">
 <title>The Power of Pomodoros</title>
 <id>urn:uuid:101a41a6-722b-4d9b-8afb-ccfb01d77499</id>
 <updated>2013-02-22T19:40:52Z</updated>
 <author><name>Elliotte Harold</name></author>
 <content>I hadn’t paid much attention to Pomodoro...</content>
</entry>
```

Cookies:

- **Definition**: Cookies are small pieces of data stored by web servers on the client-side (browser) to maintain state and track information across multiple requests.
- **Usage**: Commonly used for session management, user preferences, and tracking user behavior.

- Java provides the java.net.CookieManager and java.net.CookieStore classes to manage cookies. These classes simplify the process of sending and receiving cookies between a client and a server.

CookieManager:

- CookieManager is a class in Java's java.net package designed to handle the storage and management of cookies for HTTP clients.
- It acts as a central point for managing cookies received from and sent to web servers.
- It works together with CookieStore to manage the actual storage of cookies and CookiePolicy to determine which cookies should be accepted or rejected.

Initialization:

```
CookieManager cookieManager = new CookieManager();
CookieHandler.setDefault(cookieManager);
```

Setting Policies:

- CookiePolicy.ACCEPT_ALL: Accept all cookies.
- CookiePolicy.ACCEPT_NONE: Reject all cookies.
- CookiePolicy.ACCEPT_ORIGINAL_SERVER: Only accept cookies from the original server.

Example: cookieManager.setCookiePolicy(CookiePolicy.ACCEPT_ALL);

CookieStore:

- CookieStore is an interface in Java's java.net package that represents storage for cookies.
- It works in conjunction with CookieManager to handle the storage and retrieval of cookies.

- The CookieStore interface allows different implementations to define how cookies are stored, whether in memory, on disk, or in a database.
- The default implementation provided by Java is InMemoryCookieStore, which stores cookies in memory. This means cookies are lost when the application terminates.

Basic Operations:

1. Add a Cookie: Adds a cookie to the store.

```
URI uri = new URI("http://www.example.com");
HttpCookie cookie = new HttpCookie("name", "value");
cookieStore.add(uri, cookie);
```

2. Retrieve Cookies: Retrieves cookies associated with a specific URI or all cookies in the store.

```
// Get cookies for a specific URI
List<HttpCookie> cookiesForUri = cookieStore.get(uri);
for (HttpCookie cookie : cookiesForUri) {
    System.out.println(cookie);
}

// Get all cookies
List<HttpCookie> allCookies = cookieStore.getCookies();
for (HttpCookie cookie : allCookies) {
    System.out.println(cookie);
}
```

3. Remove Cookies: Removes a specific cookie or all cookies.

```
// Remove a specific cookie
cookieStore.remove(uri, cookie);

// Remove all cookies
cookieStore.removeAll();
```

Example: Setting up and Using CookieStore with CookieManager:

```
import java.net.CookieManager;
import java.net.CookieStore;
import java.net.HttpCookie;
import java.net.URI;
import java.net.URL;
import java.net.URLConnection;
import java.util.List;
public class CookieStoreExample {
    public static void main(String[] args) {
        try {
            // Initialize CookieManager and set it as the default CookieHandler
            CookieManager cookieManager = new CookieManager();
            CookieHandler.setDefault(cookieManager);

            // Retrieve the CookieStore from the CookieManager
            CookieStore cookieStore = cookieManager.getCookieStore();

            // Add a cookie to the store
            URI uri = new URI("http://www.example.com");
            HttpCookie cookie = new HttpCookie("username", "john_doe");
            cookieStore.add(uri, cookie);
```

```
            // Send a request to the server
            URL url = new URL("http://www.example.com");
            URLConnection connection = url.openConnection();
            connection.getContent();

            // Retrieve and print cookies from the store
            List<HttpCookie> cookies = cookieStore.getCookies();
            for (HttpCookie retrievedCookie : cookies) {
                System.out.println("Cookie: " + retrievedCookie);
            }

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```