

# **Unit 3: URLs and URIs**

By

Khushbu Kumari Sarraf

EMBA,BE

# URL

- A URL (Uniform Resource Locator) is a unique identifier used to locate a resource on the Internet. It is also referred to as a web address. URLs consist of multiple parts -- including a protocol and domain name -- that tell a web browser how and where to retrieve a resource.
- In theory, each valid URL points to a unique resource. Such resources can be an HTML page, a CSS document, an image, etc. In practice, there are some exceptions, the most common being a URL pointing to a resource that no longer exists or that has moved.

# URL

- The URL contains the name of the protocol needed to access a resource, as well as a resource name. The first part of a URL identifies what protocol to use as the primary access medium. The second part identifies the IP address or domain name -- and possibly subdomain -- where the resource is located.
- URL protocols include HTTP (Hypertext Transfer Protocol) and HTTPS (HTTP Secure) for web resources, mail to for email addresses, FTP for files on a File Transfer Protocol (FTP) server, and telnet for a session to access remote computers. Most URL protocols are followed by a colon and two forward slashes; "mail to" is followed only by a colon.
  - Optionally, after the domain, a URL can also specify:
  - a path to a specific page or file within a domain;
  - a network port to use to make the connection;
  - a specific reference point within a file, such as a named anchor in an HTML file; and
  - a query or search parameters used -- commonly found in URLs for search results.

# URL

- **Importance of a URL design**

- URLs can only be sent over the Internet using the [ASCII](#) character-set. Because URLs often contain non-ASCII characters, the URL must be converted into a valid ASCII format. URL encoding replaces unsafe ASCII characters with a "%" followed by two hexadecimal digits. URLs cannot contain spaces.

- **URL examples**

- When designing URLs, there are different theories about how to make the syntax most usable for readers and archivists. For example, in the URL's path, dates, authors, and topics can be included in a section referred to as the "slug." Consider, for example, the URL for this definition:

- ***<https://www.techtarget.com/searchnetworking/definition/URL>***

# URL

- **Basic Structure of URL**



- Look past the protocol (identified as HTTPS) and the permalink (www.techtarget.com) and we see the file path includes two paths (search networking and definition) and the title of the definition (URL).
- Additionally, some URL designers choose to put the date of the post, typically, as (YYYY/MM/DD).

# Parts of URL

- Using the URL <https://whatis.techtarget.com/search/query?q=URL> as an example, components of a URL can include:
- **The protocol or scheme.** Used to access a resource on the internet. Protocols include http, https, ftps, mailto and file. The resource is reached through the domain name system ([DNS](#)) name. In this example, the protocol is https.
- **Host name or domain name.** The unique reference the represents a webpage. For this example, whatis.techtarget.com.
- **Port name.** Usually not visible in URLs, but necessary. Always following a colon, port 80 is the default port for web servers, but there are other options. For example, :port80.
- **Path.** A path refers to a file or location on the web server. For this example, search/query.
- **Query.** Found in the URL of [dynamic pages](#). The query consists of a question mark, followed by parameters. For this example, ?.
- **Parameters.** Pieces of information in a query string of a URL. Multiple parameters can be separated by ampersands (&). For this example, q=URL.
- **Fragment.** This is an internal page reference, which refers to a section within the webpage. It appears at the end of a URL and begins with a hashtag (#). Although not in the example above, an example could be #history in the URL <https://en.wikipedia.org/wiki/Internet#History>.

# **Relative URL**

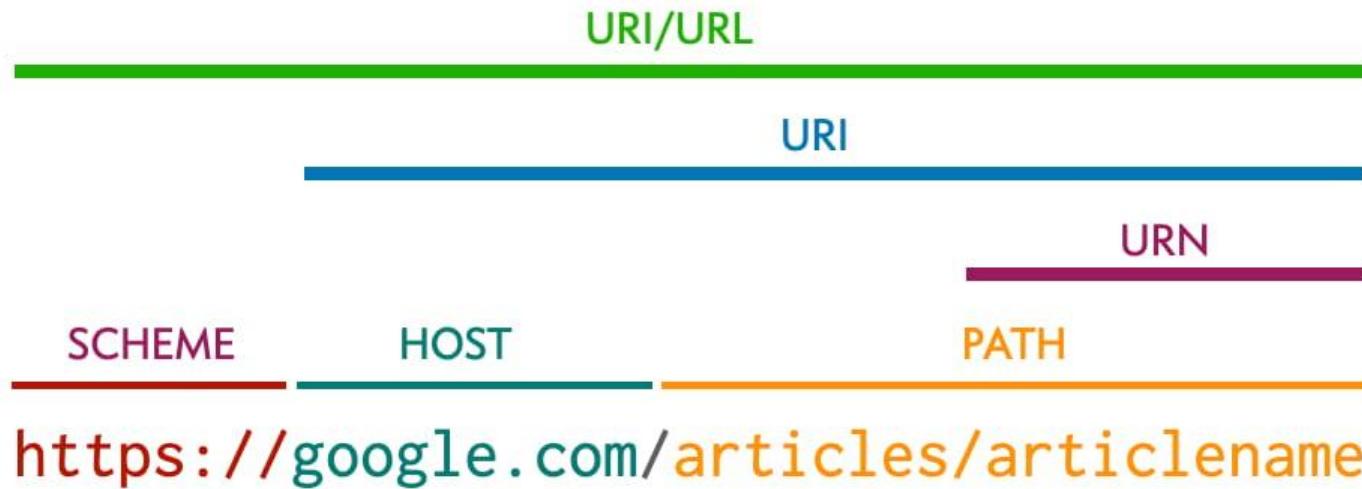
- A relative URL points to a file or directory in relation to the present file or directory. It does not include the entire pathname of the page to link but the pathname to use is relative to the current page. A relative URL is a shorthand form of absolute URL. With this shorten version of the URL, various parts of the web site address such as the protocol, domain name, and directory is inferred by the browser based on the URL of the current document or through the location specified in the `base` tag.
- It is used to point to a file that is located in the same folder and on the server as the originating file and not use them to link to pages on other domain names. An originating file is the file that contains the link. A destination file is the file to which a link is made.
- It is similar to saying, “I live in Okhla Estate, three meters from here,” which is relative to wherever “here” is but absolute way to say will be “I live 4, Okhla Estate, Delhi-110054”. Relative URLs are mostly used to link from one web page in web site to another.
- But web browser still requests the full absolute URL, not the shortened relative URL, thus browser manipulates the relative URLs into full absolute URLs.

# Relative URL

- A relative URL is a URL that only includes the path. The path is everything that comes after the domain, including the directory and slug.
- Because relative URLs don't include the entire URL structure, it is assumed that when linking a relative URL, it uses the same protocol, subdomain and domain as the page it's on.
- Here's an example of a relative URL:

/about/team/

# URI (Uniform Resource Identifier):



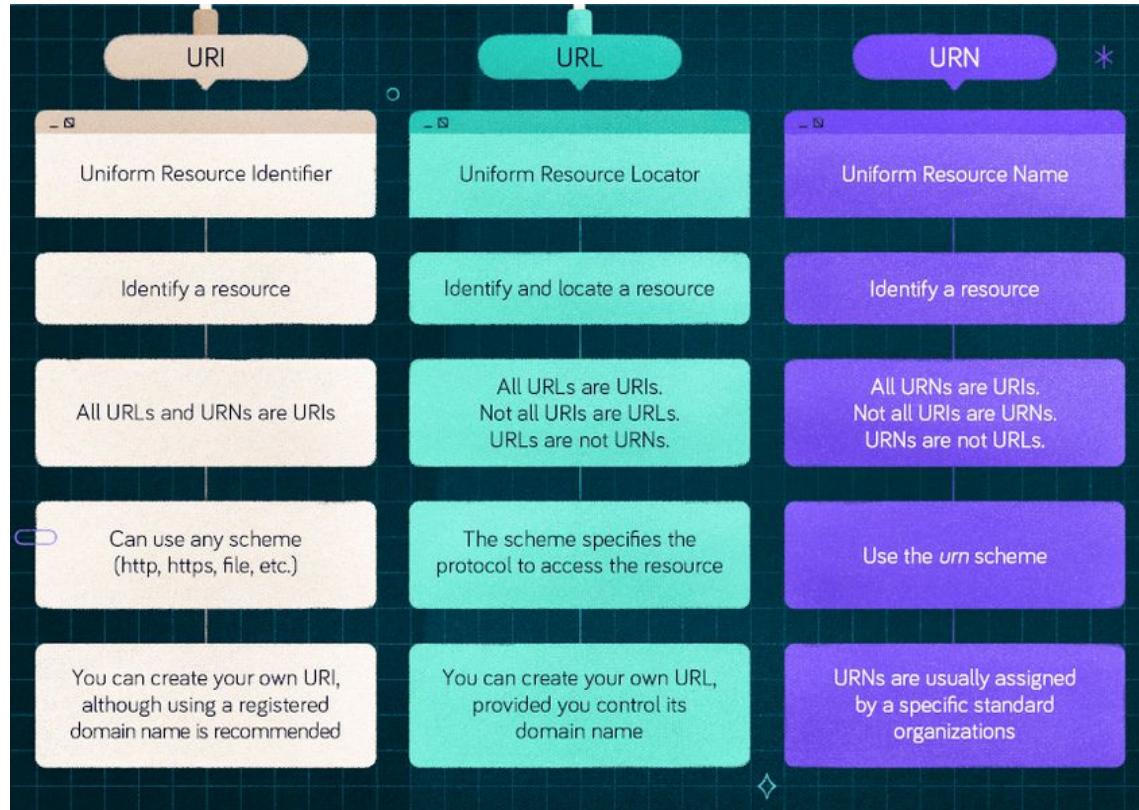
## **URI (Uniform Resource Identifier):**

- Similar to URL, URI (Uniform Resource Identifier) is also a string of characters that identifies a resource on the web either by using location, name or both. It allows uniform identification of the resources. A URI is additionally grouped as a locator, a name or both which suggests it can describe a URL, URN or both. The term identifier within the URI refers to the prominence of the resources, despite the technique used.

# Key differences between URI and URL

| URI  | URL   |
|--|---|
| URI is an acronym for Uniform Resource Identifier.   | URL is an acronym for Uniform Resource Locator.   |
| URI contains two subsets, URN, which tell the name, and URL, which tells the location.   | URL is the subset of URI, which tells the only location of the resource.                      |
| All URIs cannot be URLs, as they can tell either name or location.   | All URLs are URIs, as every URL can only contain the location.                                |
| A URI aims to identify a resource and differentiate it from other resources by using the name of the resource or location of the resource. | A URL aims to find the location or address of a resource on the web.                          |
| An example of a URI can be ISBN 0-486-35557-4.   | An example of an URL is <a href="https://www.javatpoint.com">https://www.javatpoint.com</a> . |
| It is commonly used in XML and tag library files such as JSTL and XSTL to identify the resources and binaries.                             | It is mainly used to search the webpages on the internet.                                     |
| The URI scheme can be protocol, designation, specification, or anything.   | The scheme of URL is usually a protocol such as HTTP, HTTPS, FTP, etc.                        |

# Key differences between URI, URN and URL



## **URL class in Java**

- The URL class is the gateway to any of the resource available on the internet. A Class URL represents a Uniform Resource Locator, which is a pointer to a “resource” on the World Wide Web. A resource can point to a simple file or directory, or it can refer to a more complicated object, such as a query to a database or to a search engine.
- A URL is represented by an instance of the `java.net.URL` class. A URL object manages all the component information within a URL string and provides methods for retrieving the object it identifies

## Features of URLConnection class

1. The Java URLConnection class represents a communication link between the URL and the application. It can be used to read and write data to the specified resource referred to by the URL.
2. URLConnection is an abstract class. The two subclasses HttpURLConnection and JarURLConnection make the connection between the client Java program and URL resource on the internet.
3. With the help of the URLConnection class, a user can read and write to and from any resource referenced by a URL object.
4. Once a connection is established and the Java program has a URLConnection object, we can use it to read or write or get further information like content length, etc.

# Constructors of Java URL class

| <b>URL(String spec)</b>   | Creates an instance of a URL from the String representation.  |
|---|---|
| <b>URL(String protocol, String host, int port, String file)</b> | Creates an instance of a URL from the given protocol, host, port number, and file.                        |
| <b>URL(String protocol, String host, String file)</b>           | Creates an instance of a URL from the given protocol name, host name, and file name.                      |
| <b>URL(URL context, String spec)</b>                            | Creates an instance of a URL by parsing the given spec within a specified context.                        |
| <b>URL(URL context, String spec, URLStreamHandler handler)</b>  | Creates an instance of a URL by parsing the given spec with the specified handler within a given context. |

# Commonly used methods of Java URL class

| Method                         | Description                                      |
|--------------------------------|--|
| public String getProtocol()    | it returns the protocol of the URL.              |
| public String getHost()        | it returns the host name of the URL.             |
| public String getPort()        | it returns the Port Number of the URL.           |
| public String getFile()        | it returns the file name of the URL.             |
| public String getAuthority()   | it returns the authority of the URL.             |
| public String toString()       | it returns the string representation of the URL. |
| public String getQuery()       | it returns the query string of the URL.          |
| public String getDefaultPort() | it returns the default port of the URL.          |

# Example of Java URL class

```
//URLDemo.java
import java.net.*;
public class URLEDemo{
public static void main(String[] args){
try{
URL url=new URL("http://www.javatpoint.com/java-tutorial");

System.out.println("Protocol: "+url.getProtocol());
System.out.println("Host Name: "+url.getHost());
System.out.println("Port Number: "+url.getPort());
System.out.println("File Name: "+url.getFile());

}catch(Exception e){System.out.println(e);}
}
}
```

Output:

```
Protocol: http
Host Name: www.javatpoint.com
Port Number: -1
File Name: /java-tutorial
```

# Example of Java URL class

```
//URLDemo.java  
import java.net.*;  
public class URLDemo{  
    public static void main(String[] args){  
        try{  
            URL url=new URL("https://www.google.com/search?q=javatpoint&oq=javatpoint&sourceid=chrome&ie=UTF-8");  
        }  
    }  
}
```

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

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

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

```
System.out.println("Default Port Number: " +url.getDefaultPort());
```

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

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

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

## Output:

Protocol: https

Host Name: www.google.com

Port Number: -1

Default Port Number: 443

Query String: q=javatpoint&og=javatpoint&sourceid=chrome&ie=UTF-8

Path: /search

File: /search

For more information about the study, please contact Dr. Michael J. Hwang at (319) 356-4330 or via email at [mhwang@uiowa.edu](mailto:mhwang@uiowa.edu).

1

```
}catch(Exception e){System.out.println(e);}
```

1

1

# Constructing a URL from its component parts

You can also build a URL by specifying the protocol, the hostname, and the file:

```
public URL(String protocol, String hostname, String file)  
throws MalformedURLException
```

This constructor sets the port to -1 so the default port for the protocol will be used. The `file` argument should begin with a slash and include a path, a filename, and optionally a fragment identifier. Forgetting the initial slash is a common mistake, and one that is not easy to spot. Like all `URL` constructors, it can throw a `MalformedURLException`. For example:

```
try {  
  
    URL u = new URL("http", "www.eff.org", "/blueribbon.html#intro");  
  
} catch (MalformedURLException ex)  
  
{  
  
    throw new RuntimeException("shouldn't happen; all VMs recognize http");  
}
```

# Constructing a URL from its component parts

This creates a URL object that points to **`http://www.eff.org/blueribbon.html#intro`**, using the default port for the HTTP protocol (port 80). The file specification includes a reference to a named anchor. The code catches the exception that would be thrown if the virtual machine did not support the HTTP protocol. However, this shouldn't happen in practice.

For the rare occasions when the default port isn't correct, the next constructor lets you specify the port explicitly as an `int`. The other arguments are the same. For example, this code fragment creates a URL object that points to **`http://fourier.dur.ac.uk:8000/~dma3mjh/jsci/`**, specifying port 8000 explicitly:

```
try {
    URL u = new URL("http", "fourier.dur.ac.uk", 8000, "/~dma3mjh/jsci/");
} catch (MalformedURLException ex) {
    throw new RuntimeException("shouldn't happen; all VMs recognize http"); }
```

# Constructing relative URLs

This constructor builds an absolute URL from a relative URL and a base URL:

```
public URL(URL base, String relative) throws MalformedURLException
```

For instance, you may be parsing an HTML document at <http://www.ibiblio.org/javaFAQ/index.html> and encounter a link to a file called **mailinglists.html** with no further qualifying information. In this case, you use the URL to the document that contains the link to provide the missing information. The constructor computes the new URL as <http://www.ibiblio.org/javaFAQ/mailinglists.html>. For example:

```
try {  
  
    URL u1 = new URL("http://www.ibiblio.org/javaFAQ/index.html");  
  
    URL u2 = new URL(u1, "mailinglists.html");  
  
} catch (MalformedURLException ex) {  
  
    System.err.println(ex);  
  
}
```

The filename is removed from the path of **u1** and the new filename **mailinglists.html** is appended to make **u2**. This constructor is particularly useful when you want to loop through a list of files that are all in the same directory. You can create a URL for the first file and then use this initial URL to create URL objects for the other files by substituting their filenames.

# Retrieving Data from a URL

Naked URLs aren't very exciting. What's interesting is the data contained in the documents they point to. 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. Finally, 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.

```
public final InputStream openStream() throws IOException
```

The `openStream()` method connects to the resource referenced by the URL, performs any necessary handshaking between the client and the server, and returns an `InputStream` from which data can be read. The data you get from this `InputStream` is the raw (i.e., uninterpreted) content the URL references: ASCII if you're reading an ASCII text file, raw HTML if you're reading an HTML file, binary image data if you're reading an image file, and so forth. It does not include any of the HTTP headers or any other protocol-related information. You can read from

```
try {
    URL u = new URL("http://www.lolcats.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);
}
```

## public final InputStream openStream() throws IOException

The preceding code fragment catches an `IOException`, which also catches the `MalformedURLException` that the `URL` constructor can throw, since `MalformedURLException` subclasses `IOException`. As with most network streams, reliably closing the stream takes a bit of effort. In Java 6 and earlier, we use the dispose pattern: declare the stream variable outside the `try` block, set it to null, and then close it in the `finally` block if it's not null. For example:

```
InputStream in = null
try {
    URL u = new URL("http://www.lolcats.com");
    in = u.openStream();
    int c;
    while ((c = in.read()) != -1) System.out.write(c);
} catch (IOException ex) {
    System.err.println(ex);
} finally {
    try {
        if (in != null) {
            in.close();
        }
    } catch (IOException ex) {
        // ignore
    }
}
```

# Splitting a URL into Pieces

URLs are composed of five pieces:

- The scheme, also known as the protocol
  - The authority
  - The path
  - The fragment identifier, also known as the section or ref
  - The query string
- For example, in the URL <http://www.ibiblio.org/javafaq/books/jnp/index.html?isbn=1565922069#toc>, the scheme is *http*, the authority is *www.ibiblio.org*, the path is */javafaq/books/jnp/index.html*, the fragment identifier is *toc*, and the query string is *isbn=1565922069*. However, not all URLs have all these pieces. For instance, the URL <http://www.faqs.org/rfcs/rfc3986.html> has a scheme, an authority, and a path, but no fragment identifier or query string. The authority may further be divided into the user info, the host, and the port. For
- example, in the URL <http://admin@www.blackstar.com:8080/>, the authority is *admin@www.blackstar.com:8080*. This has the user info *admin*, the host *www.blackstar.com*, and the port *8080*.
- Read-only access to these parts of a URL is provided by nine public methods: `getFile()`, `getHost()`, `getPort()`, `getProtocol()`, `getRef()`, `getQuery()`, `getPath()`, `getUserInfo()`, and `getAuthority()`.

# Splitting a URL into Pieces

## public String getProtocol()

The `getProtocol()` method returns a `String` containing the scheme of the URL (e.g., “http”, “https”, or “file”). For example, this code fragment prints **https**:

```
URL u = new URL("https://xkcd.com/727/");  
System.out.println(u.getProtocol());
```

## public String getHost()

The `getHost()` method returns a `String` containing the hostname of the URL. For example, this code fragment prints **xkcd.com**:

```
URL u = new URL("https://xkcd.com/727/");  
System.out.println(u.getHost());
```

# Splitting a URL into Pieces

## public int getPort()

The `getPort()` method returns the port number specified in the URL as an `int`. If no port was specified in the URL, `getPort()` returns -1 to signify that the URL does not specify the port explicitly, and will use the default port for the protocol. For example, if the URL is **`http://www.userfriendly.org/`**, `getPort()` returns -1; if the URL is **`http://www.userfriendly.org:80/`**, `getPort()` returns 80. The following code prints -1 for the port number because it isn't specified in the URL:

```
URL u = new URL("http://www.ncsa.illinois.edu/AboutUs/");
System.out.println("The port part of " + u + " is " + u.getPort());
```

## public int getDefaultPort()

The `getDefaultPort()` method returns the default port used for this URL's protocol when none is specified in the URL. If no default port is defined for the protocol, then `getDefaultPort()` returns -1. For example, if the URL is **`http://www.userfriendly.org/`**, `getDefaultPort()` returns 80; if the URL is **`ftp://ftp.userfriendly.org:8000/`**, `getDefaultPort()` returns 21.

# Splitting a URL into Pieces

## public String getFile()

The `getFile()` method returns a `String` that contains the path portion of a URL; remember that Java does not break a URL into separate path and file parts. Everything from the first slash (/) after the hostname until the character preceding the # sign that begins a fragment identifier is considered to be part of the file. For example:

```
URL page = this.getDocumentBase();  
  
System.out.println("This page's path is " + page.getFile());
```

If the URL does not have a file part, Java sets the file to the empty string.

## public String getPath()

The `getPath()` method is a near synonym for `getFile()`; that is, it returns a `String` containing the path and file portion of a URL. However, unlike `getFile()`, it does not include the query string in the `String` it returns, just the path.

# Splitting a URL into Pieces

## public String getRef()

The `getRef()` method returns the fragment identifier part of the URL. If the URL doesn't have a fragment identifier, the method returns `null`. In the following code, `getRef()` returns the string `xtocid1902914`:

```
URL u = new URL(  
    "http://www.ibiblio.org/javafaq/javafaq.html#xtocid1902914");  
  
System.out.println("The fragment ID of " + u + " is " + u.getRef());
```

## public String getQuery()

The `getQuery()` method returns the query string of the URL. If the URL doesn't have a query string, the method returns `null`. In the following code, `getQuery()` returns the

string `category=Piano`:

```
URL u = new URL(  
    "http://www.ibiblio.org/nywc/compositions.phtml?category=Piano");  
  
System.out.println("The query string of " + u + " is " + u.getQuery());
```

# 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.

# The URI Class

A URL object is a representation of an application layer protocol for network retrieval, whereas a URI object is purely for string parsing and manipulation. The URI class has no network retrieval capabilities. The URL class has some string parsing methods, such as getFile() and getRef(), but many of these are broken and don't always behave exactly as the relevant specifications say they should. Normally, you should use the URL class when you want to download the content at a URL and the URI class when you want to use the URL for identification rather than retrieval, for instance, to represent an XML namespace. When you need to do both, you may convert from a URI to a URL with the toURL() method, and from a URL to a URI using the toURI() method.

# 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
```

# Constructing a URI

- 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. The getRawFoo() methods return the encoded forms of the parts of the URI, while the equivalent getFoo() methods first decode any percent escaped characters and then return the decoded part:

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

# The Parts of the URI

- 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:

**public boolean isAbsolute()**

- The details of the scheme-specific part vary depending on the type of the scheme. For example, in a *tel* URL, the scheme-specific part has the syntax of a telephone number. However, in many useful URIs, including the very common *file* and *http* URLs, the scheme-specific part has a particular hierarchical format divided into an authority, a path, and a query string. The authority is further divided into user info, host, and port. The `isOpaque()` method returns false if the URI is hierarchical, true if it's not hierarchical that is, if it's opaque:

# The Parts of the URI

**public boolean isOpaque()**

- If the URI is opaque, all you can get is the scheme, scheme-specific part, and fragment identifier. However, if the URI is hierarchical, there are getter methods for all the different parts of a hierarchical URI:

```
public String getAuthority()
public String getFragment()
public String getHost()
public String getPath()
public String getPort()
public String getQuery()
public String getUserInfo()
```

## The Parts of the URI

- These methods all return the decoded parts; in other words, percent escapes, such as %3C, are changed into the characters they represent, such as <. If you want the raw, encoded parts of the URI, there are five parallel getRaw\_Foo\_() methods:

```
public String getRawAuthority()
public String getRawFragment()
public String getRawPath()
public String getRawQuery()
public String getRawUserInfo()
```

# 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);
```

# Resolving Relative URIs

- After they've executed, resolved contains the absolute URI `http://www.example.com/images/logo.png`. If the invoking URI does not contain an absolute URI itself, the resolve() method resolves as much of the URI as it can and returns a new relative URI object as a result. For example, take these three statements:

```
URI top = new URI("javafaq/books/");
URI resolved = top.resolve("jnp3/examples/07/index.html");
```

- After they've executed, resolved now contains the relative URI `javafaq/books/jnp3/examples/07/index.html` with no scheme or authority. It's also possible to reverse this procedure; that is, to go from an absolute URI to a relative one. The relativize() method creates a new URI object from the uri argument that is relative to the invoking URI. The argument is not changed. For example:

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

The URI object `relative` now contains the relative URI `images/logo.png`.

# Equality and Comparison

- URI Nodes are considered equal if they represent equal URIs, URI equality is defined in [RFC 3986](#). Primarily this requires each component of a URI to be equal on a character by character basis with some scheme specific extensions, for example the default port for a HTTP URI is 80 so that may be omitted e.g.
  - `http://example.com/test`
  - `http://example.com:80/test`
- Comparison on URIs is a lexicographical ordering by Unicode code point taking into account URI components. So they are ordered first by scheme, then by authentication details, then by host and so on. This means the ordering will group similar URIs together which is typically the desired behavior.

# Equality and Comparison

- URIs are tested for equality pretty much as you'd expect. It's not quite direct string comparison. Equal URIs must both either be hierarchical or opaque. The scheme and authority parts are compared without considering case. That is, *http* and *HTTP* are the same scheme, and *www.example.com* is the same authority as *www.EXAMPLE.com*. The rest of the URI is case sensitive, except for hexadecimal digits used to escape illegal characters. Escapes are *not* decoded before comparing. *http://www.example.com/A* and *http://www.example.com/%41* are unequal URIs.
- The `hashCode()` method is consistent with `equals`. Equal URIs do have the same hash code and unequal URIs are fairly unlikely to share the same hash code.

# Simple String Comparison

- If two URIs, considered as character strings, are identical, then it is safe to conclude that they are equivalent.
- Testing strings for equivalence requires some basic precautions. This procedure is often referred to as "bit-for-bit" or "byte-for-byte" comparision. Testing of strings for equality is normally based on pairwise comparison of the characters that make up the strings, starting from the first and proceeding until both strings are exhausted and all characters found to be equal, or a pair of characters compares unequal or one of the strings is exhausted before the other.
- These character comparisons require that each pair of characters be put in comparable form. For example, should one URI be stored in a byte array in EBCDIC encoding, and the second be in a Java String object, bit-for-bit comparisons applied naively will produce both false-positive and false-negative errors.
- Unicode defines a character as being identified by number ("codepoint") with an associated bundle of visual and other semantics. At the software level, it is not practical to compare semantic bundles, so in practical terms, character-by-character comparisons are done codepoint-by-codepoint.

## 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 example, 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. Other special characters, non alphanumeric characters, and so on, all of which may have a special meaning inside a URL or on another operating system, present similar problems. Furthermore, Unicode was not yet ubiquitous when the Web was invented, so not all systems could handle characters such as e and 本. To solve these problems, characters used in URLs must come from a fixed subset of ASCII, specifically:
  - The capital letters A–Z
  - The lowercase letters a–z
  - The digits 0–9
  - The punctuation characters - \_ . ! ~ \* ' (and ,)

## x-www-form-urlencoded

- The characters : / & ? @ # ; \$ + = and % may also be used, but only for their specified purposes. If these characters occur as part of a path or query string, they and all other characters should be encoded. The encoding is very simple. Any characters that are not ASCII numerals, letters, or the punctuation marks specified earlier are converted into bytes and each byte is written as a percent sign followed by two hexadecimal digits. Spaces are a special case because they're so common. Besides being encoded as %20, they can be encoded as a plus sign (+). The plus sign itself is encoded as %2B. The / # = & and ? characters should be encoded when they are used as part of a name, and not as a separator between parts of the URL. The URL class does not encode or decode automatically. You can construct URL objects that use illegal ASCII and non-ASCII characters and/or percent escapes. Such characters and escapes are not automatically encoded or decoded when output by methods such as getPath() and toExternalForm(). You are responsible for making sure all such characters are properly encoded in the strings used to construct a URL object. Luckily, Java provides URLEncoder and URLDecoder classes to cipher strings in this format.

# URLEncoder

- To URL encode a string, pass the string and the character set name to the URLEncoder.encode() method. For example: String encoded = URLEncoder.encode("This\*string\*has\*asterisks", "UTF-8"); URLEncoder.encode() returns a copy of the input string with a few changes. Any non alphanumeric characters are converted into % sequences (except the space, underscore, hyphen, period, and asterisk characters). It also encodes all non-ASCII characters. The space is converted into a plus sign. This method is a little overaggressive; it also converts tildes, single quotes, exclamation points, and parentheses to percent escapes, even though they don't absolutely have to be. However, this change isn't forbidden by theURL specification, so web browsers deal reasonably with these excessively encoded URLs.Although this method allows you to specify the character set, the only such character set you should ever pick is UTF-8. UTF-8 is compatible with the IRI specification, the URI class, modern web browsers, and more additional software than any other encoding you could choose.

a program that uses  
URLEncoder.encode()  
to print various  
encoded strings.

```
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");
        }
    }
}
```

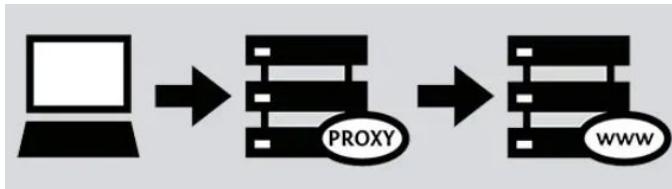
# URL Decoder

- The corresponding URLDecoder class has a static decode() method that decodes strings encoded in x-www-form-urlencoded format. That is, it converts all plus signs to spaces and all percent escapes to their corresponding character:

```
public static String decode(String s, String encoding) throws Unsupported  
Encoding Exception
```

- If you have any doubt about which encoding to use, pick UTF-8. It's more likely to be correct than anything else. An IllegalArgumentException should be thrown if the string contains a percent sign that isn't followed by two hexadecimal digits or decodes into an illegal sequence. Since URLDecoder does not touch non-escaped characters, you can pass an entire URL to it rather than splitting it into pieces first. For example:

# Proxy Server



- Basically, the proxy server plays an intermediary between the client computer and the server computer. The clients usually take the help of a proxy server for requesting any files, any web pages, or any other resources. The proxy server acts as an identification shield between the server and the client machine.
- The main and foremost feature of a proxy server is to act as a security protector device between the client computers and the server computers.
- Suppose “A” is the client computer, “B” is the server computer and “C” is the proxy server. Then when “A” requests or sends something to “B” directly, “B” can identify “A” as the sender of that request and can collect much information of “A”. But if “A” at first connect to “C” (i.e. a proxy server) and then request or sends something to “B” via “C”, then “B” can only able to collect information from “C”. In that way, “A” can able to protect and hide its personal information to “B” by taking the help of “C”.

# Proxy

## Proxy

**Proxy is hardware / software**

**Indirect access to other networks e.g INTERNET. all computers on the local network have to go through it before accessing information on the Internet.**

**Organization , universities , companies use proxy systems**

**Proxy act as gateway**

**Proxy act as Cache Server**

**Proxy share a connection to others**



# Why to use proxy

- Improve Performance
- Act as Cache server
- Filter Requests
- Prevent access to some web sites
- Browsing the WWW without any identification

# Communicating with Server-Side Programs

- The GET method is the method used by the browser to ask the server to send back a given resource: "Hey server, I want to get this resource." In this case, the browser sends an empty body. Because the body is empty if a form is sent using this method the data sent to the server is appended to the URL.
  - GET method is used to append form data to the URL in name or value pair. If you use GET, the length of the URL will remain limited. It helps users to submit the bookmark result. GET is better for the data which does not require any security or having images or word documents.
  - PHP provides two methods through which a client (browser) can send information to the server. These methods are given below, and discussed in detail:
    1. GET method
    2. POST method
  - Get and Post methods are the HTTP request methods used inside the *<form>* tag to send form data to the server.
- 
- **Features of GET**

- Here, are the important features of GET:
- It is very easy to bookmark data using GET method.
- The length restriction of GET method is limited.
- You can use this method only to retrieve data from the address bar in the browser.
- This method enables you to easily store the data.

# Communicating with Server-Side Programs

```
<html>
<body>
<form action = "gettest.php" method = "GET">
    Username: <input type = "text" name = "username" /> <br>
    Blood Group: <input type = "text" name = "bloodgroup" /> <br>
    <input type = "submit" />
</form>
</body>
</html>
```

# Communicating with Server-Side Programs

| GET   | POST  |
|---|---|
| In GET method, values are visible in the URL.   | In POST method, values are not visible in the URL.  |
| GET has a limitation on the length of the values, generally 255 characters.                                   | POST has no limitation on the length of the values since they are submitted via the body of HTTP.                   |
| GET performances are better compared to POST because of the simple nature of appending the values in the URL. | It has lower performance as compared to GET method because of time spent in including POST values in the HTTP body. |
| This method supports only string data types.  | This method supports different data types, such as string, numeric, binary, etc.                                    |
| GET results can be bookmarked.  | POST results cannot be bookmarked.  |
| GET request is often cacheable.   | The POST request is hardly cacheable.   |
| GET Parameters remain in web browser history.   | Parameters are not saved in web browser history   |

# **Web Authentication**

- Website authentication is the security process that allows users to verify their identities in order to gain access to their personal accounts on a website.
- Use different technique for authentication.

# The Authenticator Class

- The `java.net` package includes an `Authenticator` class you can use to provide a username and password for sites that protect themselves using HTTP authentication:

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

- Since `Authenticator` is an abstract class, you must subclass it. Different subclasses may retrieve the information in different ways. For example, a character mode program might just ask the user to type the username and password on `System.in`. A GUI program would likely put up a dialog box like the one shown in [Figure](#).



*An authentication dialog*

# The Authenticator Class

```
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()
```

- These methods either return the information as given in the last call to `requestPasswordAuthentication()` or return `null` if that information is not available. (If the port isn't available, `getRequestingPort()` returns -1.) The `getRequestingURL()` method returns the complete URL for which authentication has been requested—an important detail if a site uses different names and passwords for different files. The `getRequestorType()` method returns one of the two named constants (i.e., `Authenticator.RequestorType.PROXY` or `Authenticator.RequestorType.SERVER`) to indicate whether the server or the proxy server is requesting the authentication

## The PasswordAuthentication Class

- `PasswordAuthentication` is a very simple final class that supports two read-only properties: `username` and `password`. The `username` is a `String`. The `password` is a `char` array so that the password can be erased when it's no longer needed. A `String` would have to wait to be garbage collected before it could be erased, and even then it might still exist somewhere in memory on the local system, possibly even on disk if the block of memory that contained it had been swapped out to virtual memory at one point. Both `username` and `password` are set in the constructor:

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

Each is accessed via a `getter` method:

```
public String getUserName()
public char[] getPassword()
```

# The JPasswordField Class

- One useful tool for asking users for their passwords in a more or less secure fashion is the `JPasswordField` component from Swing:
  - `public class JPasswordField extends JTextField`
- This lightweight component behaves almost exactly like a text field. However, anything the user types into it is echoed as an asterisk. This way, the password is safe from anyone looking over the user's shoulder at what's being typed on the screen.  
`JPasswordField` also stores the passwords as a `char` array so that when you're done with the password you can overwrite it with zeros. It provides the `getPassword()` method to return this:
  - `public char[] getPassword()`