**What Is a URL?**

If you've been surfing the Web, you have undoubtedly heard the term URL and have used URLs to access HTML pages from the Web.

It's often easiest, although not entirely accurate, to think of a URL as the name of a file on the World Wide Web because most URLs refer to a file on some machine on the network. However, remember that URLs also can point to other resources on the network, such as database queries and command output.

**Definition:**

URL is an acronym for *Uniform Resource Locator* and is a reference (an address) to a resource on the Internet.

A URL has two main components:

* Protocol identifier: For the URL http://example.com, the protocol identifier is http.
* Resource name: For the URL http://example.com, the resource name is example.com.

Note that the protocol identifier and the resource name are separated by a colon and two forward slashes. The protocol identifier indicates the name of the protocol to be used to fetch the resource. The example uses the Hypertext Transfer Protocol (HTTP), which is typically used to serve up hypertext documents. HTTP is just one of many different protocols used to access different types of resources on the net. Other protocols include File Transfer Protocol (FTP), Gopher, File, and News.

The resource name is the complete address to the resource. The format of the resource name depends entirely on the protocol used, but for many protocols, including HTTP, the resource name contains one or more of the following components:

**Host Name**

The name of the machine on which the resource lives.

**Filename**

The pathname to the file on the machine.

**Port Number**

The port number to which to connect (typically optional).

**Reference**

A reference to a named anchor within a resource that usually identifies a specific location within a file (typically optional).

For many protocols, the host name and the filename are required, while the port number and reference are optional. For example, the resource name for an HTTP URL must specify a server on the network (Host Name) and the path to the document on that machine (Filename); it also can specify a port number and a reference.

# Creating a URL

The easiest way to create a URL object is from a String that represents the human-readable form of the URL address. This is typically the form that another person will use for a URL. In your Java program, you can use a String containing this text to create a URL object:

URL myURL = new URL("http://example.com/");

The URL object created above represents an absolute URL. An absolute URL contains all of the information necessary to reach the resource in question. You can also create URL objects from a relative URL address.

## Creating a URL Relative to Another

A relative URL contains only enough information to reach the resource relative to (or in the context of) another URL.

Relative URL specifications are often used within HTML files. For example, suppose you write an HTML file called JoesHomePage.html. Within this page, are links to other pages, PicturesOfMe.html and MyKids.html, that are on the same machine and in the same directory as JoesHomePage.html. The links to PicturesOfMe.html and MyKids.html from JoesHomePage.html could be specified just as filenames, like this:

<a href="PicturesOfMe.html">Pictures of Me</a>

<a href="MyKids.html">Pictures of My Kids</a>

These URL addresses are relative URLs. That is, the URLs are specified relative to the file in which they are contained — JoesHomePage.html.

In your Java programs, you can create a URL object from a relative URL specification. For example, suppose you know two URLs at the site example.com:

http://example.com/pages/page1.html

http://example.com/pages/page2.html

You can create URL objects for these pages relative to their common base URL: http://example.com/pages/ like this:

URL myURL = new URL("http://example.com/pages/");

URL page1URL = new URL(myURL, "page1.html");

URL page2URL = new URL(myURL, "page2.html");

This code snippet uses the URL constructor that lets you create a URL object from another URL object (the base) and a relative URL specification. The general form of this constructor is:

URL(URL baseURL, String relativeURL)

The first argument is a URL object that specifies the base of the new URL. The second argument is a String that specifies the rest of the resource name relative to the base. If baseURL is null, then this constructor treats relativeURL like an absolute URL specification. Conversely, if relativeURL is an absolute URL specification, then the constructor ignores baseURL.

This constructor is also useful for creating URL objects for named anchors (also called references) within a file. For example, suppose the page1.html file has a named anchor called BOTTOM at the bottom of the file. You can use the relative URL constructor to create a URL object for it like this:

URL page1BottomURL = new URL(page1URL,"#BOTTOM");

## Other URL Constructors

The URL class provides two additional constructors for creating a URL object. These constructors are useful when you are working with URLs, such as HTTP URLs, that have host name, filename, port number, and reference components in the resource name portion of the URL. These two constructors are useful when you do not have a String containing the complete URL specification, but you do know various components of the URL.

For example, suppose you design a network browsing panel similar to a file browsing panel that allows users to choose the protocol, host name, port number, and filename. You can construct a URL from the panel's components. The first constructor creates a URL object from a protocol, host name, and filename. The following code snippet creates a URL to the page1.html file at the example.com site:

new URL("http", "example.com", "/pages/page1.html");

This is equivalent to

new URL("http://example.com/pages/page1.html");

The first argument is the protocol, the second is the host name, and the last is the pathname of the file. Note that the filename contains a forward slash at the beginning. This indicates that the filename is specified from the root of the host.

The final URL constructor adds the port number to the list of arguments used in the previous constructor:

URL gamelan = new URL("http", "example.com", 80, "pages/page1.html");

This creates a URL object for the following URL:

http://example.com:80/pages/page1.html

If you construct a URL object using one of these constructors, you can get a String containing the complete URL address by using the URL object's toString method or the equivalent toExternalForm method.

## URL addresses with Special characters

Some URL addresses contain special characters, for example the space character. Like this:

http://example.com/hello world/

To make these characters legal they need to be encoded before passing them to the URL constructor.

URL url = new URL("http://example.com/hello%20world");

Encoding the special character(s) in this example is easy as there is only one character that needs encoding, but for URL addresses that have several of these characters or if you are unsure when writing your code what URL addresses you will need to access, you can use the multi-argument constructors of the [java.net.URI](https://docs.oracle.com/javase/8/docs/api/java/net/URI.html" \t "_blank) class to automatically take care of the encoding for you.

URI uri = new URI("http", "example.com", "/hello world/", "");

And then convert the URI to a URL.

URL url = uri.toURL();

## MalformedURLException

Each of the four URL constructors throws a MalformedURLException if the arguments to the constructor refer to a null or unknown protocol. Typically, you want to catch and handle this exception by embedding your URL constructor statements in a try/catch pair, like this:

try {

URL myURL = new URL(...);

}

catch (MalformedURLException e) {

// exception handler code here

// ...

}

See [Exceptions](https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html) for information about handling exceptions.

**Note:**

URLs are "write-once" objects. Once you've created a URL object, you cannot change any of its attributes (protocol, host name, filename, or port number).

**Parsing a URL**

The URL class provides several methods that let you query URL objects. You can get the protocol, authority, host name, port number, path, query, filename, and reference from a URL using these accessor methods:

**getProtocol**

Returns the protocol identifier component of the URL.

**getAuthority**

Returns the authority component of the URL.

**getHost**

Returns the host name component of the URL.

**getPort**

Returns the port number component of the URL. The getPort method returns an integer that is the port number. If the port is not set, getPort returns -1.

**getPath**

Returns the path component of this URL.

**getQuery**

Returns the query component of this URL.

**getFile**

Returns the filename component of the URL. The getFile method returns the same as getPath, plus the concatenation of the value of getQuery, if any.

**getRef**

Returns the reference component of the URL.

**Note:**

Remember that not all URL addresses contain these components. The URL class provides these methods because HTTP URLs do contain these components and are perhaps the most commonly used URLs. The URL class is somewhat HTTP-centric.

You can use these get*XXX* methods to get information about the URL regardless of the constructor that you used to create the URL object.

The URL class, along with these accessor methods, frees you from ever having to parse URLs again! Given any string specification of a URL, just create a new URL object and call any of the accessor methods for the information you need. This small example program creates a URL from a string specification and then uses the URL object's accessor methods to parse the URL:

import java.net.\*;

import java.io.\*;

public class ParseURL {

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

URL aURL = new URL("http://example.com:80/docs/books/tutorial"

+ "/index.html?name=networking#DOWNLOADING");

System.out.println("protocol = " + aURL.getProtocol());

System.out.println("authority = " + aURL.getAuthority());

System.out.println("host = " + aURL.getHost());

System.out.println("port = " + aURL.getPort());

System.out.println("path = " + aURL.getPath());

System.out.println("query = " + aURL.getQuery());

System.out.println("filename = " + aURL.getFile());

System.out.println("ref = " + aURL.getRef());

}

}

Here is the output displayed by the program:

protocol = http

authority = example.com:80

host = example.com

port = 80

path = /docs/books/tutorial/index.html

query = name=networking

filename = /docs/books/tutorial/index.html?name=networking

ref = DOWNLOADING

# Reading Directly from a URL

After you've successfully created a URL, you can call the URL's openStream() method to get a stream from which you can read the contents of the URL. The openStream() method returns a [java.io.InputStream](https://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html" \t "_blank) object, so reading from a URL is as easy as reading from an input stream.

The following small Java program uses openStream() to get an input stream on the URL http://www.oracle.com/. It then opens a BufferedReader on the input stream and reads from the BufferedReader thereby reading from the URL. Everything read is copied to the standard output stream:

import java.net.\*;

import java.io.\*;

public class URLReader {

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

URL oracle = new URL("http://www.oracle.com/");

BufferedReader in = new BufferedReader(

new InputStreamReader(oracle.openStream()));

String inputLine;

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

System.out.println(inputLine);

in.close();

}

}

When you run the program, you should see, scrolling by in your command window, the HTML commands and textual content from the HTML file located at http://www.oracle.com/. Alternatively, the program might hang or you might see an exception stack trace. If either of the latter two events occurs, you may have to [set the proxy host](https://docs.oracle.com/javase/tutorial/networking/urls/_setProxy.html) so that the program can find the Oracle server.

# Connecting to a URL

After you've successfully created a URL object, you can call the URL object's openConnection method to get a URLConnection object, or one of its protocol specific subclasses, e.g. [java.net.HttpURLConnection](https://docs.oracle.com/javase/8/docs/api/java/net/HttpURLConnection.html" \t "_blank)

You can use this URLConnection object to setup parameters and general request properties that you may need before connecting. Connection to the remote object represented by the URL is only initiated when the URLConnection.connect method is called. When you do this you are initializing a communication link between your Java program and the URL over the network. For example, the following code opens a connection to the site example.com:

try {

URL myURL = new URL("http://example.com/");

URLConnection myURLConnection = myURL.openConnection();

myURLConnection.connect();

}

catch (MalformedURLException e) {

// new URL() failed

// ...

}

catch (IOException e) {

// openConnection() failed

// ...

}

A new URLConnection object is created every time by calling the openConnection method of the protocol handler for this URL.

You are not always required to explicitly call the connect method to initiate the connection. Operations that depend on being connected, like getInputStream, getOutputStream, etc, will implicitly perform the connection, if necessary.

Now that you've successfully connected to your URL, you can use the URLConnection object to perform actions such as reading from or writing to the connection. The next section shows you how.

# Reading from and Writing to a URLConnection

The URLConnection class contains many methods that let you communicate with the URL over the network. URLConnection is an HTTP-centric class; that is, many of its methods are useful only when you are working with HTTP URLs. However, most URL protocols allow you to read from and write to the connection. This section describes both functions.

## Reading from a URLConnection

The following program performs the same function as the URLReader program shown in [Reading Directly from a URL](https://docs.oracle.com/javase/tutorial/networking/urls/readingURL.html).

However, rather than getting an input stream directly from the URL, this program explicitly retrieves a URLConnection object and gets an input stream from the connection. The connection is opened implicitly by calling getInputStream. Then, like URLReader, this program creates a BufferedReader on the input stream and reads from it. The bold statements highlight the differences between this example and the previous:

import java.net.\*;

import java.io.\*;

public class **URLConnectionReader** {

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

URL oracle = new URL("http://www.oracle.com/");

**URLConnection yc = oracle.openConnection();**

BufferedReader in = new BufferedReader(new InputStreamReader(

**yc.getInputStream()**));

String inputLine;

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

System.out.println(inputLine);

in.close();

}

}

The output from this program is identical to the output from the program that opens a stream directly from the URL. You can use either way to read from a URL. However, reading from a URLConnection instead of reading directly from a URL might be more useful. This is because you can use the URLConnection object for other tasks (like writing to the URL) at the same time.

Again, if the program hangs or you see an error message, you may have to set the proxy host so that the program can find the Oracle server.

## Writing to a URLConnection

Many HTML pages contain forms — text fields and other GUI objects that let you enter data to send to the server. After you type in the required information and initiate the query by clicking a button, your Web browser writes the data to the URL over the network. At the other end the server receives the data, processes it, and then sends you a response, usually in the form of a new HTML page.

Many of these HTML forms use the HTTP POST METHOD to send data to the server. Thus writing to a URL is often called posting to a URL. The server recognizes the POST request and reads the data sent from the client.

For a Java program to interact with a server-side process it simply must be able to write to a URL, thus providing data to the server. It can do this by following these steps:

1. Create a URL.
2. Retrieve the URLConnection object.
3. Set output capability on the URLConnection.
4. Open a connection to the resource.
5. Get an output stream from the connection.
6. Write to the output stream.
7. Close the output stream.

Here is a small servlet named [ReverseServlet](https://docs.oracle.com/javase/tutorial/networking/urls/examples/ReverseServlet.java" \t "_blank) (or if you prefer a [cgi-bin](https://docs.oracle.com/javase/tutorial/networking/urls/examples/backwards) script). You can use this servlet to test the following example program.

The servlet running in a container reads from its InputStream, reverses the string, and writes it to its OutputStream. The servlet requires input of the form string=string\_to\_reverse, where string\_to\_reverse is the string whose characters you want displayed in reverse order.

Here's an example program that runs the ReverseServlet over the network through a URLConnection:

import java.io.\*;

import java.net.\*;

public class Reverse {

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

if (args.length != 2) {

System.err.println("Usage: java Reverse "

+ "http://<location of your servlet/script>"

+ " string\_to\_reverse");

System.exit(1);

}

String stringToReverse = URLEncoder.encode(args[1], "UTF-8");

URL url = new URL(args[0]);

URLConnection connection = url.openConnection();

connection.setDoOutput(true);

OutputStreamWriter out = new OutputStreamWriter(

connection.getOutputStream());

out.write("string=" + stringToReverse);

out.close();

BufferedReader in = new BufferedReader(

new InputStreamReader(

connection.getInputStream()));

String decodedString;

while ((decodedString = in.readLine()) != null) {

System.out.println(decodedString);

}

in.close();

}

}

Let's examine the program and see how it works. First, the program processes its command-line arguments:

if (args.length != 2) {

System.err.println("Usage: java Reverse "

+ "http://<location of your servlet/script>"

+ " string\_to\_reverse");

System.exit(1);

}

String stringToReverse = URLEncoder.encode(args[1], "UTF-8");

These statements ensure that the user provides two and only two command-line arguments to the program. The command-line arguments are the location of the ReverseServlet and the string that will be reversed. It may contain spaces or other non-alphanumeric characters. These characters must be encoded because the string is processed on its way to the server. The URLEncoder class methods encode the characters.

Next, the program creates the URL object, and sets the connection so that it can write to it:

URL url = new URL(args[0]);

URLConnection connection = url.openConnection();

connection.setDoOutput(true);

The program then creates an output stream on the connection and opens an OutputStreamWriter on it:

OutputStreamWriter out = new OutputStreamWriter(connection.getOutputStream());

If the URL does not support output, getOutputStream method throws an UnknownServiceException. If the URL does support output, then this method returns an output stream that is connected to the input stream of the URL on the server side — the client's output is the server's input.

Next, the program writes the required information to the output stream and closes the stream:

out.write("string=" + stringToReverse);

out.close();

This code writes to the output stream using the write method. So you can see that writing data to a URL is as easy as writing data to a stream. The data written to the output stream on the client side is the input for the servlet on the server side. The Reverse program constructs the input in the form required by the script by prepending string= to the encoded string to be reversed.

The servlet reads the information you write, performs a reverse operation on the string value, and then sends this back to you. You now need to read the string the server has sent back. The Reverse program does it like this:

BufferedReader in = new BufferedReader(

new InputStreamReader(

connection.getInputStream()));

String decodedString;

while ((decodedString = in.readLine()) != null) {

System.out.println(decodedString);

}

in.close();

If your ReverseServlet is located at http://example.com/servlet/ReverseServlet, then when you run the Reverse program using

http://example.com/servlet/ReverseServlet "Reverse Me"

as the argument (including the double quote marks), you should see this output:

Reverse Me

reversed is:

eM esreveR