CYCLE 4: URL, URL Connection Classes

Java.net.URL class is an abstraction of a Uniform Resource Locator

Constructors

There are six construtors. All these construtors throw a MalformedURLException if we try to create a URL for an **unsupported protocol.** Exactly which protocols are supported are implementation dependent i.e. they vary from one version java to another. Other than verifying that it recognizes the protocol part of the URL, Java does not make any checks about correctness of the URLs it constructs. The programmer is resposible for making sure that the URLs created are valid. For instance, Java does not check that the hostname in an HTTP URL does not contain spaces. It does not check that a mailto URL actually contain an email address. Java does not check the URL to make sure that it points at an existing host or that it meets any other requirements for URLs.

Constructing a URL from a string

public URL(String url) throws MalformedURLException

Constructing a URL from its component parts

public URL(String protocol, String hostname, String file) throws MalfomedURLException

This constructor sets the port to -1 so the default port for the protocol will be used. The *file* argument should begin a slash, and include a path, a filename and optionally a reference to a named anchor.

Constructing a URL by specifing a port also

public URL(String protocol, String host, int port, String file) throws MalformedURLException

If default port is not used by any server for a protocol,we use this constructor.

Constructing relative URLs

public URL(URL base,String relative) throws MalformedURLException

When we are parsing an HTML document at *http://metalab.unc.edu/javafaq/index.html* and encounter a link to a file called *mailinglist.html* with no further qualifying information. In this case we use the URL to the document that contains the link to provide the missing information.

For example:

```
try{    URL u1=new URL("http://metalab.unc.edu/javafaq/index.html");

        URL u2=new URL(u1,"mailinglist.html");
} catch(MalformedURLException e){System.err.println(e);}
```

The filename is removed from thepath 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. We 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. We can also use this constructor when we want to create a URL relative to the applet's document base or codebase which we retrieve using the getDocumentBase() or getCodeBase() methods of the java.applet.Applet class.

Example:

```
import java.net.*;
import  java.applet.*;
import java.awt.*;
public class RelativeURLTest extends Applet{
public void init()
{
try{
URL base=this.getDocumentBase();
URL relative=new URL(base,"mailinglists.html");
this.setLayout(new GridLayout(2,1));
this.add(new Label(base.toString());
this.add(new Label(relative.toString());
}
catch(MalformedURLException e){ this.add(new Label("This shouldn't happen"));}
}
}
```

Constructors for specifing URL Stream Handlers

Java 1.2 adds two URL constructors that allow us to specify the protocol hadler used for the URL. The first constructor builds a relative URL from a base URL and a relative part, then uses the specified handler to do the work for the URL. The seconds builds the URL from its component pieces, then uses the specified handler to do the work for the URL.

public URL(URL base, String relative, URLStreamHandler handler) throws MalformedURLException

public URL(String protocol, String host, int port, String file, URLStreamHandler) throws MalformedURLException

All URL objects have URLStreamHandler objects to do their work for them. These two constructors allow you to change from the default URLStreamHandler to the one of our own choosing. This is useful for working with URLs whose schemes aren't supported in a particular virtual machine as well as for adding functionality that the default stream handler doesn't provide, like asking the user for a username and password.

Splitting a URL into pieces

URLs can be thought of as composed of five pieces

- The scheme also known as the protocol
- The authority
- The path
- The ref, also known as the section or named anchor
- The query string

Example: *http://metalab.unc.edu/javafaq/books/jnp/index.html?isbn=1565922069#toc*

- scheme: http
- authority : metalab.unc.edu
- path: /javafaq/books/jnp/index.html
- ref: toc
- query string: isbn=1565922069

Read only access to these parts of the URL are provided by five methods. getFile(), getHost(), getPort(), getProtocol(), getRef(). Java 1.3 adds four more methods getQuery(), getPath(), getUserInfo(), getAuthority().

public String getProtocol()

It returns a String containing the scheme of the URL. Ex: http,https,file

public String getHost()

It returns a String containing the hostname of the URL. The host string is not necessarily a valid hostname or address. In particular URLs that incorporate usernames like *ftp://anonymous:anonymous@wuarchive.wustl.edu/* include the user info in the host i.e. getHost() returns anonymous:anonymous@wuarchive.wustl.edu not simply wuarchive.wustl.edu. For reasons of backward compatibility Java 1.3 did not change the semantics of getHost() method to return only the host rather than the host plus the user info.

public int getPort()

It return the port number specified in the URL as an int. If no port is specified in the URL, the getPort() returns -1 to signify that the URL does not specify the port explicitly, and will use the default port for the protocol.

public String getFile()

It returns a String that contains the path and file portion of a URL. Everything from first / after the hostname until the character preceding the # sign that begins a section is considered to be part of the file.

public String getPath()

It is a synonym for getFile(). The reason for this duplicate method is to sync up Java's terminology with the URI specification in RFC 2396. RFC 2396 calls what we and Java have been calling the "file" the "path" instead. The getFile() method isn't deprecated as of Java 1.3 but it may become so in future releases.

public String getRef()

It returns the named anchor part of the URL. If the URL doen't have a named anchor, the method return null.

public String getQuery()

It return the query String of the URL. If the URL doesn't have a query string it returns null.

public String getUserInfo()

Some URLs have usernames and occasionally password information. This comes after the scheme and before the host. An @ symbol delimits it. This method returns both of them. If an URL does not contain user information, this method returns null. In the URL like *mailto:elharo@metalab.unc.edu, elharo@metalab.unc.edu* is a path not user info and the host. This URL specifies the remote recipient of the message rather than the username and host that's sending the message.

public String getAuthority()

Between the scheme and the path of a URL we will find the authority. It indicates the authority that is resolving the resource. This method returns the authority as it exists in the URL, with or without the user info and port.

Retrieving data from a URL

Two important methods available in the URL class to retrieve data from it are:
public final InputStream openStream() throws IOException
public URLConnection openConnection() throws IOException

1. public final InputStream openStream() throws IOException

It connects to the resource referenced by the URL, performs any necessary handshaking between the client and the server and then returns an InputStream from which data can be read. The data we get from this InputStream is raw(i.e. uninterpreted) contents of the file the URL references. ASCII if we reading an ASCII text file, raw HTML if we reading an HTML

file, binary image data if we reading an image file and so forth. It does include any HTTP headers or any other protocol-related information.

```
try{
        URL u=new URL("http://www.hamsterdance.com");
        InputStream in=u.openStream();
        int c;
        while((c=in.read())!=-1) System.out.write(c);
    }
Catch(IOException e){System.out.println(e);}
```

2. public URLConnection openConnection() throws IOException

The openConnection() method opens a socket to the specified URL and returns a URLConnection object. A URLConnection represents an open connection to a network resource. If the call fails, openConnection() throws an IOException.
This method is used when we want to communicate directly with the server. The URLConnection gives us access to everything sent by the server: in addition to the document itself, in its raw form (i.e. HTML, plain text, binary image data), we can access all the headers used by the protocol in use. For example if we are retrieving an HTML document, the URLConnection will let us access the HTTP headers as well as the raw HTML.

URLConnection

A program that uses the URLConnection class directly follows this basic sequence of steps:

1. Construct a URL object.
2. Invoke the URL object's openConnection() method to retrieve a URLConnection object for that URL.
3. Configure the URLConnection.
4. Read the header fields.
5. Get an input stream and read data.
6. Get an output stream and write data.
7. Close the connection.

Reading the Header

HTTP servers provide a substantial amount of information in the MIME headers that precede each response. For example, here's a typical MIME header returned by an Apache web server running on Solaris.

```
HTTP 1.1 200 OK
Date: Mon, 18 Oct 1999 20:06:48 GMT
Server: Apache/1.3.4 (Unix) PHP/3.0.6 mod_perl/1.17
Last-Modified: Mon, 18 Oct 1999 12:58:21 GMT
ETag: "1e05f2-89bb-380b19cd"
```

Accept-Ranges: bytes
Content-Length: 35259
Connection: close
Content-Type: text/html

Retrieving Specific MIME Header Fields
Common fields from the MIME header can be retrieved using the following methods:

public String getContentType()

This method returns the MIME content type of the data. It relies on the web server to send a proper MIME header, including the content type. It throws no exceptions and returns null if the content type is not available. text/html will be by far the most common content type we will encounter when connecting to web servers. Other commonly used types include text/plain, image/gif, image/jpeg.

public int getContentLength()

This method tells us the number of bytes that in the content. Many servers send content-length headers only when transferring a binary file, not when transferring a text file. If there is no content-length header, it returns -1.It is used when we need to know exactly how many bytes to read when we need to create a buffer large enough to hold the data in advance.

public String getContentEncoding()

This method returns a String that tells us how the content is encoded. If the content is sent unencode (as is commonly with the HTTP servers) then this method returns null. It throws no exceptions. The most commonly used content encoding on the Web is probably x-gzip, which can be straightforwardly decoded using a *java.util.zip.GzipInputStream.*

public long getDate()

This method returns a long that tells us when the document was sent, milliseconds since midnight, GMT, January 1, 1970. It can be converted to a *java.util.Date.*
Date dc=new Date(uc.getDate());
This is the time the document was sent as seen from the server. It may not agree with the time on our machine. If the MIME header does not include a Date header, getDate returns 0.

public long getExpiration()

Some documents have server-based expiration dates that indicate when the document should be deleted from the cache and reloaded from the server. It return a long indicating the number of milliseconds after 12.00 A.M., GMT January 1, 1970, at which the document expires. If

the MIME header does not include an Expiration header, it returns 0 which means the document does not need to be expired, and can remain in the cache indefinitely.

public long getLastModified()

It returns the date on which the document was last modified. Again the date is given as the numbet of milliseconds since midnight, GMT, January 1, 1970. If the MIME header does not include a Last-modified header this method returns 0.