Chapter 8

In this chapter we study the features of Java that allows us to work directly at the URL (Universal Resource Locator) and we are particularly interested in HTTP (Hyper Text Transfer Protocol). An URL is a string representing a resource accessible via the Internet. This resource can be an email address, a text file, html file or a program to be run remotely.

Different types of resources are specified by their **scheme**, each different pattern is implemented by a specific protocol:

• ***http*** is a scheme implemented by the HTTP protocol,

• ***ftp*** is a scheme implemented by the FTP protocol,

• ***mailto*** scheme is implemented by the SMTP protocol

• ***telnet*** is a scheme implemented by the TELNET protocol.

The general syntax of a URL consists of the schema name and resource name:

```
<schema>:<ressource du schema>
```

The resource schema depends on the protocol used. For HTTP this is the server name and path to the http file *. on this server.

For FTP, it is necessary to provide the user name, the password, the FTP server name and port number (21 by default).

Examples:

```
http://www.ireste.fr/fdl/vcl/index.html
ftp://pbakowsk:motdepasse@serveur:21
mailto:ythomas@ireste.fr
```

java.net.URL class

In Java, the java.net.URL class contains the functions needed to develop an application using URL. We have several constructors that may be used to create an object of the URL class,. All URL constructors raise the exception MalformedURLException.

```
public URL(String url) throws MalformedURLException
public URL(String baseURL, String url) throws MalformedURLException
public URL(String scheme, String baseURL, String url) throws MalformedURLException
```

Examples:

```
public URL(“http://www.ireste.fr/fdl”) throws MalformedURLException
public URL(“http://www.ireste.fr” ,”/fdl”) throws MalformedURLException
public URL(“http”,“www.ireste.fr”, ”/fdl”) throws MalformedURLException
```

Reading the characteristics of an URL

The URL class provides several methods to get the characteristics of an URL:

- *getProtocol()* - renvoie l'identificateur du protocole,
- *getHost()* - renvoie l'identificateur de la machine hôte,
- *getPort()* - renvoie le numéro du port,
- *getFile()* - renvoie le chemin d'accès au fichier,
- *getRef()* - renvoie la référence interne dans le fichier.

Example:

```
import java.net.*;
import java.io.*;
public class ParseURL {
   public static void main(String[] args) throws Exception {
      URL aURL = new URL("http://java.sun.com:80/docs/books/"
                     + "tutorial/index.html#DOWNLOADING");
      System.out.println("protocole = " + aURL.getProtocol());
      System.out.println("hote = " + aURL.getHost());
      System.out.println("chemin d'access = " + aURL.getFile());
      System.out.println("port = " + aURL.getPort());
      System.out.println("ref = " + aURL.getRef());
   }
}
```

Below is the result of executing this program:

```
protocole = http
hote = java.sun.com
chemin d'access = /docs/books/tutorial/index.html
port = 80
ref = DOWNLOADING
```

The following program is an applet that scrutinizes the implementation of various schemes on a server site. Note that the URL constructor uses three arguments: *scheme*, *host* and *path* .

```
import java.applet.*;
import java.awt.*;
import java.net.*;

public class ProtocolTesterApplet extends Applet {
TextArea ta = new TextArea();

public void init() {
   this.setLayout(new BorderLayout(20,20));
   this.add("Center",ta);
}
```

```
public void start() {
String hote = "www.ireste.fr";
String chemin = "/fdl";
String[] schema = {"http", "https", "ftp", "mailto", "telnet", "file", "rmi","finger", "daytime","nfs"};
for(int i=0; i<schema.length;i++) {
try {
  URL u = new URL(schema[i],hote,chemin);
  ta.append(schema[i] + " est implemente\r\n");
  }
  catch(MalformedURLException e) {
    ta.append(schema[i] + " not implemented\r\n"); }
  }
}
}
```

The result of the execution is displayed in a window type *TextArea*:

```
http est implemente
https est implemente
ftp est implemente
mailto est implemente
telnet est implemente
file est implemente
rmi n'est pas implemente
finger n'est pas implemente
daytime n'est pas implemente
nfs est implemente
```

Reading in an URL

When an URL object is created, it can open its content directly through the method *openStream()* and read the input stream by the method *read()*.

```
try {
URL u = new URL("http://www.google.fr");
InputStream in = u.openStream();
int car;
while ((car = in.read()) != -1)  System.out.write(car);
}
catch (IOException e)
 {  System.err.println(e); }
```

The following example is more complete. The *openStream()*method opens a *java.io.InputStream*. stream The stream is then processed by the classes *InputStreamReader* and *BufferedReader* to read a string.

```
import java.net.*;
import java.io.*;
public class URLReader {
   public static void main(String[] args) throws Exception {
            URL google = new URL("http://www.google.fr/");
   BufferedReader in = new BufferedReader( new InputStreamReader( google.openStream()));
            String inputLine;
          while ((inputLine = in.readLine()) != null)
             System.out.println(inputLine);
          in.close();
 }
}
```

URL connections

The URLConnection class provides communication control with an HTTP server. Through an URL connection, the user can parse MIME headers associated with HTTP transfer and can send its own data by *POST* and *PUT* actions.

Using a connection usually goes through several stages:

- construction of an URL object,
- opening a connection *openConnection ()* on Object URL
- configuring the connection,
- reading the header,
- reading data
- writing data
- closing the connection.

Some of these stages are optional in the case of reading the header or writing data. Reading and writing can be executed in reverse order: writing and reading.

Below is an example of an URL connection:

```
try {
  URL u = new URL(“http://www.google.fr”);
  URLConnection uc = u.openConnection();
  }
  catch (MalformedURLCOnnection e)
                { System.err.println(e); }
  catch (IOException e)
                { System.err.println(e);}
```

Reading from an URL

The following program opens an URL connection and gets an input stream of bytes to put in the buffer:

```
import java.net.*;
import java.io.*;
public class URLConnectionReader1 {
public static void main(String[] args) throws Exception {
  if(args.length>0)
  {
       URL u = new URL(args[0]);
       URLConnection uc = u.openConnection();
       InputStream rawin = uc.getInputStream();
                  InputStream tampon = new BufferedInputStream(rawin);
       Reader r = new InputStreamReader(tampon);
            int c;
       while ((c = r.read()) != -1) System.out.println((char)c);
       in.close();
    }
}
```

The next version reads a string line by line:

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

public class URLConnectionReader2 {
public static void main(String[] args) throws Exception {
  if(args.length>0)  {
       URL u = new URL(args[0]);
       URLConnection uc = u.openConnection();
       BufferedReader in = new BufferedReader(new InputStreamReader(uc.getInputStream()));
       String inputLine;
       while ((inputLine = in.readLine()) != null)
          System.out.println(inputLine);
       in.close();
    }
  }
}
```

Reading MIME header

For each transfer of information, the server generates an HTTP header which contains the characteristics of information sent.

Example of a header:

```
HTTP 1.1  200 OK    // version an error code
Date: Mon, 5 Nov 2000 12:05:32 GMT
```

```
Server: Apache/1.3.4 (Linux)
Last-Modified: Mon, 5 Nov 2000 09:12:41 GMT
ETag: “5e06f3-74aa- 320a357f”
Accept-Ranges: bytes
Content-Length: 23871
Connection: close
Content-Type: text/html
```

The characteristics of the header can be retrieved by the get methods. For example the type of content can be obtained by the method *getContentType();* size information via the method *getContentLenght().*

The following program is looking for a website and extracts a file (*args[0]).* The MIME settings of the file are displayed on the screen. The file itself is stored in the working directory.

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


public class URLSaveFile {
public static void main(String[] args) {
for(int i=0; i<args.length;i++) {
try {  URL racine = new URL(args[0]);     saveFile(racine);
 }
 catch(MalformedURLException e) { System.err.println(args[i] + " is not an URL"); }
 catch(IOException e) { System.err.println(e); }
 }
}
public static void saveFile(URL u) throws IOException{
URLConnection uc = u.openConnection();
            String contentType = uc.getContentType();
            int taille = uc.getContentLength();
            System.out.println("Type: " + contentType);
            System.out.println("Taille: " + taille);
            System.out.println("Date :" + new Date(uc.getDate()));
            System.out.println("Modification :" + new Date(uc.getLastModified()));
            System.out.println("Expiration :" + new Date(uc.getExpiration()));
            System.out.println("Encodage :" + uc.getContentEncoding());
            if(taille == -1 ) { throw new IOException("pas de fichier"); }
InputStream rawin = uc.getInputStream();
            InputStream tampon = new BufferedInputStream(rawin);
            byte[] data = new byte[taille];
            int bytesRead = 0;
            int offset =0;
            while(offset <taille) {
              bytesRead = rawin.read(data,offset,data.length-offset);
```

```
                if(bytesRead == -1) break;
                offset +=bytesRead;
                }
rawin.close();
            if(offset != taille) {
              throw new IOException("lus" + offset + "demandes" + taille); }
            String nomfichier = u.getFile();
            nomfichier = nomfichier.substring(nomfichier.lastIndexOf('/') + 1);
            FileOutputStream fout = new FileOutputStream(nomfichier);
            fout.write(data);
            fout.flush();
            fout.close();
            }
}
```

Access to protected pages

Access to several Web sites requires interaction between the user and the server. For example reading a page protected by a username and password requires user interaction.

The authentication mechanism is based on an *Authentication Manager*, subject to a subclass of abstract class *Authenticator*. Its static method *setDefaut()* can install such a manager.

After installing the manager, the objects of the URL class using its services each time a query returns a status code 401 *Unathorized* or 407 *Proxy Authentication Required*.

The object URL makes a call to the static method *requestPasswordAuthentication()* of the *Authenticator* class. This method makes a call to *getPasswordAuthentication()* method of Authentication Manager and returns a "user-ID and a password. These results are encapsulated in an object of class *PasswordAuthentication* and placed by the URL class in the header of access authorization.

```
import java.net.*;
import java.io.*;
public class AuthenticatorTest extends Authenticator {
private String name;
private String passwd;
public AuthenticatorTest(String name, String passwd) {  // storing user-ID and  password
this.name=name;
this.passwd=passwd;
}
public PasswordAuthentication getPasswordAuthentication() {  // called if returned status: 401 or 407
System.out.println("Site: " + getRequestingSite());
System.out.println("Port: " + getRequestingPort());
System.out.println("Scheme: " + getRequestingScheme());
System.out.println("Prompt: " + getRequestingPrompt());
```

```
// retourns le user-ID and password
return new PasswordAuthentication(name,passwd.toCharArray());
 }
public static void main(String[] args) throws Exception {
if(args.length!=1) {
  System.err.println("Usage: java AuthenticatorTest <url>");
  System.exit(1);
  }
AuthenticatorTest auth;   // creation of authentification manager
auth = new AuthenticatorTest("bako","123");
Authenticator.setDefault(auth);
URL url = new URL(args[0]);
URLConnection uc = url.openConnection();
InputStream in = uc.getInputStream();
int c;
 while((c=in.read())!=-1){
  System.out.print((char)c);
  }
 }
}
```

Local cache

Most communication with the server go through the Web Cache installed locally on the client machine. The *setUseCaches()* method is used to validate or invalidate the cache usage.

Exemple:

```
try {
URL u = new URL("http://www.serveur.fr/fichier.htm");
URLConnection uc = u.openConnection();
if(uc.getUseCaches())              // utilisation of cache autorised
 {
 uc.setUseCaches(false);           // utilisation of cache forbidden
  }
// lire directement dans l'URL
 }
catch (IOException e) { System.err.println(e); }
```

Sending the data to server

GET* or *POST

Programs that use an URL connection may send their data to the server. The connection can be configured for GET or POST actions.

The data sent by the GET action are attached to the URL, their size is usually quite limited, for example 1024 bytes. To send the data of more consistent size, use the POST action.

The *setDoOutput(false)* method involves using the GET action; *setDoOutput(true)* changes the action GET to POST.

Exemple:

```
try {
URL u = new URL("http://www.serveurcommunicant.fr");
URLConnection uc = u.openConnection();
if(!uc.getDoOutput())              // test if the GET is set to false
      {
      uc.setDoOutput(true);       // connection for action POST
      }
// write to the connection by POST
 }
catch (IOException e) { System.err.println(e); }
```

Of a connection is configured for the action POST, the data can be prepared in an output buffer by: *BufferedInputStream* or *BufferedWriter*.

Example:

```
try {
URL u = new URL("http://www.serveurcommunicant.fr");
URLConnection uc = u.openConnection();
if(!uc.getDoOutput())              // test if the GET is set to false
 {  uc.setDoOutput(true);       // connection for action POST   }
 OutputStream raw = uc.getOutputStream();
 OutputStream tampon = new BufferedOutputStream(raw);
OutputStreamWriter out = new OutputStreamWriter(tampon,"8859_1");
out.write("un=Jean&tp=sockets&exemple=url@connexion\r\n");
out.flush();
out.close();
 }
catch (IOException e) { System.err.println(e); }
```

Note: The Java program stores data in the buffer until the closure of the connection by *close()*. This is necessary to determine the parameter *Content-length* in http header.

The message is preceded by the header:

```
POST /cgi-bin/resultat.pl  HTTP 1.0
Content-type: application/x-www-form-urlencoded
Content-length: 42
un=Jean&tp=sockets&exemple=url%40connexion        // format encode
```

The following program sends a string to http//java.sun.com. The program to execute is housed in the /cgi-bin/ and called backwards.pl.

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


public class Reverse {
   public static void main(String[] args) throws Exception {
            if (args.length != 1) {
              System.err.println("Utilisation:  java Reverse " + "string to return ");
              System.exit(1);}
            String stringToReverse = URLEncoder.encode(args[0]);         // encoding
            URL url = new URL("http://java.sun.com/cgi-bin/backwards.pl");
            URLConnection connection = url.openConnection();
            connection.setDoOutput(true);          // initialisation de l'action POST
            PrintWriter out = new PrintWriter(connection.getOutputStream());
            out.println("string=" + stringToReverse);
            out.close();
            BufferedReader in = new BufferedReader(new InputStreamReader(connection.getInputStream()));
            String inputLine;
            while ((inputLine = in.readLine()) != null)
               System.out.println(inputLine);
            in.close();
   }
}
```

Class loader

In the previous examples we used connections to get URL pages (files) of data. The JVM can also read the remote codebyte classes and execute them locally. To perform this operation you must use a class loader.

The class loader installed by default searches for classes locally in the directories and archives. In this chapter we focus on the classe java.net.URLClassLoader dedicated to the loading of remote classes available on the Internet.

The simplest form of a constructor of the class loader is *URLClassLoader(URL [] urls).* URL is given as an argument, the user can provide a reference http or a local file.

Examples:

```
URLClassLoader(http://www.ireste.fr/fdl/test/)
URLClassLoader(file:///home/test/)
```

The base of an URL used by the class loader are considered as directory if it ends with a slash and ('/'), as Java archives in all other cases.

For a given object loader, the reception of a class is done via a call to the *loadClass()* with the argument representing the name of the requested class. This method inherits from *ClassLoader*. If the requested class is not found from the URL base, the loader charger throws ClassNotFoundException.

We will prepare two classes to be loaded remotely. The first will be registered on the local disk (scheme: *file*), the second on the WEB server (scheme: *http*). Both classes implement the same interface Chargeable.

```
public interface Chargeable {              // a compiler avant la classe SimpleClassLoader
public void charge();
{
public class FichierLocal implements Chargeable {         // class to load
public void charge() {                                     // method charge()
System.out.println("Ici un fichier local"); }
}
public class FichierWEB implements Chargeable {
public void charge() {                                    // method charge()
System.out.println("Ici un fichier WEB"); }
}
```

Ces deux classes peuvent être dynamiquement chargées à partir d'un URL indiqué dans la ligne de commande du programme ci-dessous. Ce programme utilise un chargeur *URLClassLoader()*.

```
import java.net.*;
public class SimpleClassLoader {
public static void main(String[] args) throws Exception {
if(args.length<1) {
   System.err.println("Utilisation: java SimpleClassLoader <classe>");
    System.exit(1); }
URL[] urls = new URL[] {
new URL("http://www.ireste.fr/fdl/ars/test/"),
new URL("file:///home/pbakowsk/test/")  };
URLClassLoader cl = new URLClassLoader(urls);          // creation of loader chargeur
Class c = null;  // declaration of the class to load
try {
  c = cl.loadClass(args[0]);  // chargement de la classe
  Chargeable ic = (Chargeable)c.newInstance();
```

```
  ic.charge();
   } catch(ClassNotFoundException e) { System.err.println("Pas de classe " + args[0]);}
   } catch(ClassCastException e) { System.err.println("La classe " + args[0] + " no Chargeable"); }
 } }
```

Looking for a resource

The class *URLClasseLoader* contains the method *getLocalResource(String name).* This method allows you to search the resource whose name is passed as parameter. The method *getLocalResource(String name)* returns null if there is no URL in the base that does allow access to a resource of that name.

Warning: The method *findRessources()* requires the full name of the resource that is not necessarily a bytecode of a class.

Example:

```
import java.net.*;
import java.util.Enumeration;
public class ChercheRessources {
public static void main(String[] args) throws Exception {
if(args.length<1) {
   System.err.println("Utilisation: java ChercheRessources <nomres>");
   System.exit(1); }
URL[] urls = new URL[] {
            new URL("http://www.ireste.fr/"),          // scheme http
            new URL("file:///c:\\"),                   // scheme file
            new URL("ftp://ftp.inria.fr/"),            // scheme ftp  };
URLClassLoader cl = new URLClassLoader(urls);
System.out.println("The rssource " + args[0] + " exists at the following URL :");
Enumeration e = cl.findResources(args[0]);
while(e.hasMoreElements()) {
   System.out.println("\t"+((URL) e.nextElement()).toString()); }
 }
}
```

Summary

In this chapter we studied the URL class and different methods to communicate with web servers. Depending on the initialization mode, this communication may be uni-or bi-directional. The information flowing between the client and the server can be very different in nature: text file, binary file (images, sounds). The use of MIME headers allows us to characterize the size, nature and other characteristics of the transmitted information.

The Java class files are very specific. Their reading and remote loading require the intervention (methods) of the java.net.URLClassLoader class.