EDA095
Uniform Resource Locators (URLs) and HTML Pages

Pierre Nugues

Lund University
http://cs.lth.se/pierre_nugues/

March 30, 2016

Covers: Chapter 5, *Java Network Programming*, 4th ed., Elliotte Rusty Harold

The Web

Everybody knows what the web is. No need to present it.
When it started, it was a combination of three main features:

- A document format in plain text: HTML derived from the older and clumsy SGML
- A communication protocol to transfer data: HTTP
- A way to name and address objects on the network: URLs

We will review ways to handle URLs and HTML directly from Java.

Uniform Resource Locators (URL)

A URL is the name of a "resource" on the Internet and an access mode.
It goes beyond data that can be exchanged using HTTP.
It can extend to FTP, telnet, mail protocols, and many more.
In its simplest form, a URL is a string that consists of three parts:
`protocol://hostname/path/object`

- The `protocol` can be `http`, `ftp`, `telnet`, `rmi`, etc.
- The `hostname` is an Internet address such as `cs.lth.se` or `130.235.16.34`
- The `path/object` corresponds for instance to `/home/pierre/file.html`

URLs are defined by RFC 2396 and RFC 2732.
(`http://www.rfc-editor.org/rfc/rfc2396.txt`, `http://www.rfc-editor.org/rfc/rfc2732.txt`)

URL (Continued)

Protocols have a default port.
The URL can specify a new one:
`http://cs.lth.se:80/EDA095/index.html`
A URL can be absolute or relative to a base URL
The link `labs.html` in a document whose URL is
`http://cs.lth.se:80/EDA095/index.html`
has the URL
`http://cs.lth.se:80/EDA095/labs.html`

The URL Class

The URL class defines a way to build and parse URL strings
Constructors:

- `URL(String spec)`
- `URL(String protocol, String host, String file)`
- `URL(URL context, String spec)`

The last one creates a URL by parsing the given spec within a specified context.
`http://docs.oracle.com/javase/8/docs/api/java/net/URL.html`
The constructor throws a `MalformedURLException` if it fails to create the URL

A First Program Using the URL Class

Creating URLs

The constructor does not check the syntax. See MyDoc3

```
import java.net.*;
public class MyURL {
  public static void main(String[] args) {
    try {
      URL myDoc1 = new URL("http://www.cs.lth.se/index.html");
      URL myDoc2 = new URL("http", "www.cs.lth.se", "/index.html");
      URL myDoc3 = new URL("http", "www.cs.lth.se", "index.html");
      URL myDoc4 = new URL(new URL("http://www.cs.lth.se/EDA095/"),
        "labs.shtml");
      URL myDoc5 = new URL(new URL("http://www.cs.lth.se/EDA095"),
        "labs.shtml");
    } catch (Exception e) { }
  }
} //MyURL.java
```

Client and Server Communications Using HTTP

The communication between a server and a client in its simplest form starts with a TCP connection from the client to port 80.
Then we have the exchange:

- The client sends a request
- The server sends a response.

Both request and response messages feature a header that consists of parameter-value pairs
In addition:

- The request consists of a command word (HTTP method) to identify the request, parameters, and possibly other data.
- The response consists of a response code and objects to be displayed or rendered by the client

HTTP Request

The request behind the URL `http://cs.lth.se/pierre_nugues` consists of:

1. HTTP method, URL, version

   ```
   GET /pierre_nugues HTTP/1.1
   ```

2. Sequence of parameter names (46 types) followed by ':' and values – pairs `Name: Value`

   ```
   Accept: text/plain
     ...
   Host: cs.lth.se
   User-Agent: Mozilla/4.0
   ```

3. Empty line: `\r\n`
4. Possibly a message body (data) whose size is given by the `Content-Length` attribute

RFC 2616 (`http://www.ietf.org/rfc/rfc2616.txt`)

HTTP Response

Servers send a response: header followed by data

1. Protocol, status code, textual phrase

   ```
   HTTP/1.0 200 OK
   ```

2. Sequence of parameter names followed by ':' and values

   ```
   Date: Wed, 28 Mar 2007 12:12:54 GMT
   Server: Apache/2.0.52 (sparc-sun-solaris2.8)
   ...
   Connection: close
   ```

3. Empty line: `\r\n`

4. Data

   ```
   <html>
   ...
   </html>
   ```

Getting Data from a Server

In its simplest form, the client does not send data
It just requests the content of the URL file
For HTTP, the corresponding methods are `GET` or `POST`.
The method is sent automatically by the URL class and is hidden to the programmer.
We will review the steps to get data from simple to more complex
The URL class uses `GET` by default, hides the header details, and receives data

Getting Data from a Server

The `URL` class has methods to open a connection from a URL:

- `InputStream openStream()` //Opens a connection and returns an `InputStream` for reading from that connection.
- `URLConnection openConnection()` //Returns a `URLConnection` object.

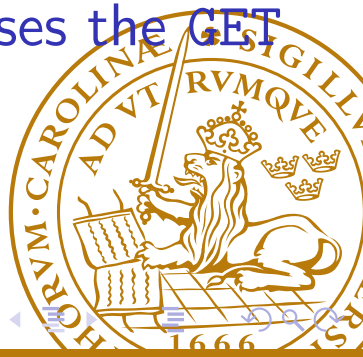
The opening methods will connect to the resource using the specified protocol

Once open, the stream can be read and contain data sent by the server

The protocol machinery (http, ftp, etc.) is invisible to the programmer

We will first consider examples with `openStream()`

The combination of the `URL` constructor and `openStream()` uses the `GET` method



Getting Data Using openStream()

```
//ViewHTML.java
try {
  URL myDoc = new URL("http://cs.lth.se/");
  InputStream is = myDoc.openStream();
  BufferedReader bReader =
    new BufferedReader(new InputStreamReader(is));
  String line;
  while ((line = bReader.readLine()) != null) {
    System.out.println(line);
  }
} catch (Exception e) {
  e.printStackTrace();
}
```

Relative URLs

Using the 3rd constructor, we can create URLs relatively to a context (MyURL2.java)

```
URL myDoc1 = new URL("http://www.cs.lth.se/home/index.html");
URL myDoc2 = new URL(myDoc1, "pierre.html");
URL myDoc3 = new URL(myDoc1, "/pierre.html");
URL myDoc4 = new URL(myDoc1, "http://www.lu.se/pierre.html");
URL myDoc5 = new URL(myDoc1, "mailto:pierre@cs.lth.se");
```

URL Exceptions

The description in the URL class exceptions

> *Throws:* `MalformedURLException` – *If the string specifies an unknown protocol.*

is not consistent with that of `MalformedURLException`:

> *Thrown to indicate that a malformed URL has occurred. Either no legal protocol could be found in a specification string or the string could not be parsed.*

The URL constructor only checks the protocol.

Exceptions

```
//MyURL3.java
package url;
import java.net.*;
public class MyURL {
  public static void main(String[] args) {
    try {
      URL myDoc1 = new URL("http://www.cs.lth.se/index.html");
      URL myDoc2 = new URL("http", "www.cs.lth.se", "index.html");
      URL myDoc3 =
        new URL(new URL("http://www.cs.lth.se/EDA095/"),
          "labs.shtml");
      URL myDoc4 = new URL("http:///www.cs;lth.se\\^/index.html");
    } catch (Exception e) {
      e.printStackTrace();
    }
```

Exceptions II

```
    try {
      URL myDoc5 = new URL("htp:////www.cs;lth.se\\^/index.html ");
    } catch (Exception e) {
      e.printStackTrace();
    }
    try {
      URL myDoc6 = new URL("htp", "www.cs.lth,se", "index.html");
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Methods of the URL Class

Some useful methods are:

- `String getProtocol()` //Gets the protocol name.
- `String getHost()` //Gets the host name.
- `int getPort()` //Gets the port number.
- `String getPath()` //Gets the path part and the file name.
- `String getFile()` //Gets the path, the file name, and the query if it exists. In fact nearly the same as `getFile()`, see documentation

Using URL Methods

```
//myURL4.java
URL myDoc1 = new URL("http://www.cs.lth.se/index.html");
URL myDoc2 = new URL("http", "www.cs.lth.se", "/index.html");
URL myDoc3 = new URL("http:///www.cs;lth.se\\^/index.html");
System.out.println("Protocols: "+ myDoc2.getProtocol() + " " +
  myDoc3.getProtocol());
System.out.println("Files: "+ myDoc2.getFile() + " " +
  myDoc3.getFile());
System.out.println("Ports: "+ myDoc2.getPort() + " " +
  myDoc3.getPort());
System.out.println("Hosts: "+ myDoc2.getHost() + " " +
  myDoc3.getHost());
```

Testing Available Protocols

Using the URL class, can we find protocols supported by a machine
Idea: Create a set of URL objects and call `getProtocol()`
Try: http, https, ftp, tftp, mailto, telnet, file, gopher, ladp, jar, nfs, jdbc, rmi, etc.
and catch the errors...
This is the idea of ProtocolTester.java, Elliotte Rusty Harold, *Java Network Programming*, page 186.
The code available here:
`http://www.cafeaulait.org/books/jnp3/examples/index.html`

Uniform Resource Identifiers (URI)

URIs are name conventions close to URLs (RFC 2396)
`[scheme:]scheme-specific-part[#fragment]`
However, URIs do not provide a method to access a network resource.
URIs can be absolute (with a scheme) or relative (without a scheme)
URIs can also be:

- Opaque: The scheme-specific part does not begin with a `/`. They are not parsed as: `mailto:java-net@java.sun.com`, `news:comp.lang.java`, `urn:isbn:096139210x`
- Hierarchical: They are absolute and begin with a `/` or they are relative and have no scheme as: `http://java.sun.com/j2se/1.3/` or `docs/guide/collections/designfaq.html#28`

Parsing a URI

A hierarchical URI is parsed according to the syntax

`[scheme:][//authority][path][?query][#fragment]`

If the authority is a server, its syntax is:

`[user-info@]host[:port]`

The user info can consist of a user name and a password, `anonymous:pierre@cs.lth.se` on an anonymous ftp server

URI Methods

The Java implementation of the `URI` class is more conformant than the URL one.
The URI class has methods to build a URI from its components.
Constructors:

- `URI(String str)`
- `URI(String scheme, String ssp, String fragment)`
- `URI(String scheme, String userInfo, String host, int port, String path, String query, String fragment)`

It has Booleans to determine whether it is:

- opaque or hierarchical, `isOpaque()`
- relative or absolute, `isAbslolute()`

URI Methods

The URI class has methods to parse a string (a hierarchical URI):

`[scheme:][//authority][path][?query][#fragment]`

- `String getScheme()`
- `String getAuthority()`
- `String getPath()`
- `String getQuery()`
- `String getFragment()`
- `URI parseServerAuthority()`, parses the authority, (`String getUserInfo()`, `String getHost()`, `int getPort()`)

URISplitter.java from Elliotte Rusty Harold, Java Network Programming, page 218.

```
java url.URISplitter http://www.cs.lth.se:80/pierre/index.html
```

Sending Data from the HTTP Client

Clients can send data to HTTP servers using a list of parameter-value pairs:

- book=Java Network Programming
- author=Harold

This is used when you fill in HTML forms, for instance
`GET` and `POST` are the two methods to carry this out
We review `GET` now

Sending Parameters with GET

GET sends the list of parameter-value pairs in the URL in the query part:

```
[scheme:][//authority][path][?query][#fragment]
```

The parameter list must comply with a specific format and encoding for the accents:

```
Arg1=Value1&Arg2=Value2
```

as in

```
book=Java+Network+Programming&author=Harold
```

The preferred encoding is UTF-8
The URLEncoder class carries this out with the method:

```
static String encode(String s, String enc)
```

HTML Forms

Web clients send data to servers using HTML forms
Forms define the possible parameters that will be filled by the user with values
Input components are defined by `<input>` elements with a type attribute.
Possible types are hidden, text (default), radio, scroll down menus, submit buttons, reset, etc.
Data in the form of pairs (parameter name, value) are sent using GET and POST methods.
GET uses the URL string to send the parameters
POST sends a MIME message

Example of Search Box

The course home page: `http://www.eit.lth.se/`, contains an input box:

```
<input name="search" type="hidden" value="1" />
  <input name="L" type="hidden" value="0" />
  <input name="q" type="text" value="" class="field" /> 
  <input name="start" type="hidden" value="1" />
  <input name="no_cache" type="hidden" value="1" />
  <input name="x" type="submit" value="Sök" class="button" />
  <input type="hidden" name="category" value="" />
</form>
```

Queries

The query Nugues to the course home page is a sequence of pairs (name, value)

```
http://www.eit.lth.se/index.php?search=1&L=0&q=Nugues&
  start=1&no_cache=1&x=S%F6k&category=
```

`URISplitter` extracts the query:

```
search=1&L=0&q=Nugues&start=1&no_cache=1&x=S?k&category=
```

We can create a `GET` request using the URL constructor and send it to LTH using `openStream()`
POST would send

```
Content-Type: application/x-www-form-urlencoded
Content-Length: 42
Connection: close


s=1&so=1&i=sv&category=cs&q=nugues&x=S%F6k
```

Encoding Characters in URLs

Some characters are encoded like *Sök* in our example
From the Java specifications:

- The alphanumeric characters a through z, A through Z and 0 through 9 remain the same.
- The special characters ., -, *, and _ remain the same.
- The space character ' ' is converted into a plus sign +.
- All other characters are unsafe and are first converted into one or more bytes using some encoding scheme. Then each byte is represented by the 3-character string %xy, where xy is the two-digit hexadecimal representation of the byte. The recommended encoding scheme to use is UTF-8.

You have to encode the string explicitly with UTF-8
We will review UTF-8 and other types of encoding in a next lecture

Encoding Strings

To demonstrate `URLEncoder`, we will use the program EncodeTest.java from Elliotte Rusty Harold, *Java Network Programming*, page 210.
We call:

```
URLEncoder.encode("Sök", "UTF-8");
URLEncoder.encode("Sök", "ISO-8859-1");
URLEncoder.encode("This string has spaces", "UTF-8");
```

In the textbook, there is a mistake with the encoding option of javac
It assumes that the editor saves the program in UTF-8.
Thus is not always the case. Many editors use the machine's encoding, MacRoman on a Mac, Latin 1 on Linux...
Use:

```
pierre% javac -encoding UTF8 url/EncodeTest.java
```

only if your program has been saved with UTF 8.

URL Decoders

URLDecoder decodes a string encoded using UTF-8 in the machine's encoding
It has one method:

```
static String decode(String s, String enc)
```

You have to decode the string explicitly with UTF-8

```
System.out.println(URLDecoder.decode("S%C3%B6k", "UTF-8"));
System.out.println(URLDecoder.decode("S%F6k", "ISO-8859-1"));
System.out.println(URLDecoder.decode("S%C3%B6k", "ISO-8859-1"));
System.out.println(URLDecoder.decode("S%F6k", "UTF-8"));
```

(DecodeTest.java)