Introduction to Java for C++ Programmers

URI's and URL's

By: Mahboob Ali

High Level API's

- No ports and sockets.

- URI's (Universal Resource Identifier)
- URL's (Universal Resource Locator).

URI

- While working with *java.net* package,
  - URI is an identifier to identify a resource either by location, or a name or both.
  - URI can specify a *relative path*. (public://someimage.jpg)
  - URL is an identifier that includes information about how to access the resource it identifies.
  - URL has to be an *absolute path*. (http://mysite/images/someimage.jpg)
  - Recommendation: keep using URI until you actually want to access a resource, and then converts the URI into URL.

Low-level API vs High-level API

- Low-level API used the following classes: *Socket*, *ServerSocket*, and *DatagramSocket*.

- High-level API used the following classes: *URI*, *URL*, *URLConnection*, and *HttpURLConnection*.

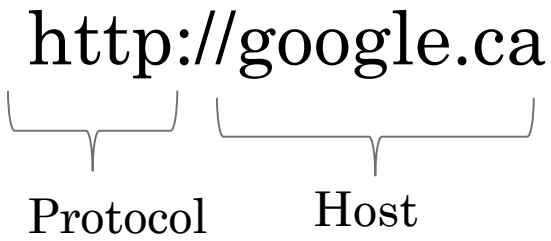
- URI contains eight components:
  - Scheme (http, ftp, etc)
  - Authority (three parts)
    - User-info (user name)
    - Host (www.example.com)
    - Port (123)
  - Path (/images)
  - Query (?=what+is)
  - Fragment (#)

URL (Uniform Resource Locator)

- It is a reference to a source on the Internet.
- By providing the URL's to your favorite Web browser it locate files on the internet, similarly when you provide address on your letter so that post office can locate the correspondent.
- URL is a URI but a URI is not a URL.

URL Components

- URL has two main components.



- The host name contain one or more of the following components,
  - Host name
  - File name
  - Port number
  - Reference

Creating URL

URL myURL = new URL(“http://google.ca”)

Absolute URL: contains all of the information necessary to reach the resource like above.

Relative URL: A relative URL contains only enough information to reach the resource relative to (or in the context of) another URL. Like a page contain links to other pages.

Parsing a URL

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, returns -1 otherwise
getPath()	Returns the path component of this URL
getQuery()	Returns the query component of this URL
getFile()	Returns the filename component of the URL
getRef()	Returns the reference component of the URL

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

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

        URL aURL = new
URL("https://ict.senecacollege.ca/course/jav745?q=course/jav745");

        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());
    }
}
```

URL OpenStream

- URL's `openStream()` method to get a stream from which you can read the contents of the URL.

- openStream() method returns a java.io.InputStream object

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

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

	URL seneca = new
	URL(" https://ict.senecacollege.ca/course/jav745?q=course/jav745 ");

	BufferedReader in = new BufferedReader( new InputStreamReader(seneca.openStream()));

	String inputLine;

	while ((inputLine = in.readLine()) != null)
		System.out.println(inputLine);
	in.close();
	}
}
```

URL Connection

```
import java.net.*;
import java.io.*;
public class ReadSites {
        public static void main(String[] args) {
              for (int i = 0; i < args.length; i++) {
                    try {
                  URL u = new URL(args[i]);

                  URLConnection connn = u.openConnection();
        InputStreamReader isr = new InputStreamReader(connn.getInputStream());
                  BufferedReader br = new BufferedReader(isr);
                  String s;
                  while ((s = br.readLine()) != null)
                           System.out.println(s);
                  } catch (MalformedURLException e) {
                           System.err.println(e);
                  } catch (IOException e) {
                            System.err.println(e); }
                } }       }
```

- When you do this you are initializing a communication link between your Java program and the URL over the network.
- Use of URLConnection object to perform actions such as reading from or writing to the connection.

Sockets

- **Definition:** A *socket* is one endpoint of a two-way communication link between two programs running on the network.

- Java Socket programming can be connection-oriented (TCP) or connection-less (UDP).

Why Socket Programming?

- URL's and URLConnection's provide high-level mechanism for accessing resources on internet.
- Socket programming provides low-level network communication.
- Like writing *client-server* applications.

Client Server Application

- The server provides some service like performing database queries or sending out current stock prices.
- The client uses the service provided by the server.
- The communication that occurs between the client and the server must be reliable.
- No data can be dropped.
- Data must arrive on the client side in the same order in which the server sent it.

What is Socket?

- Server runs on a specific computer and has a socket that is bound to a specific port number.
- The server just waits, listening to the socket for a client to make a connection request.
- The client knows the hostname of the machine on which the server is running and the port number on which the server is listening.

- To make connection,
  - Client exchange information with server.
  - Once client identification is done.
  - Local port is also assigned for communication during connection.

- If all goes well,
  - server accepts the connection.
  - server gets a new socket bound to the same local port.
  - its remote endpoint set to the address and port of the client

- Why server needed new socket?

- So that it can continue to listen to the original socket for connection requests.

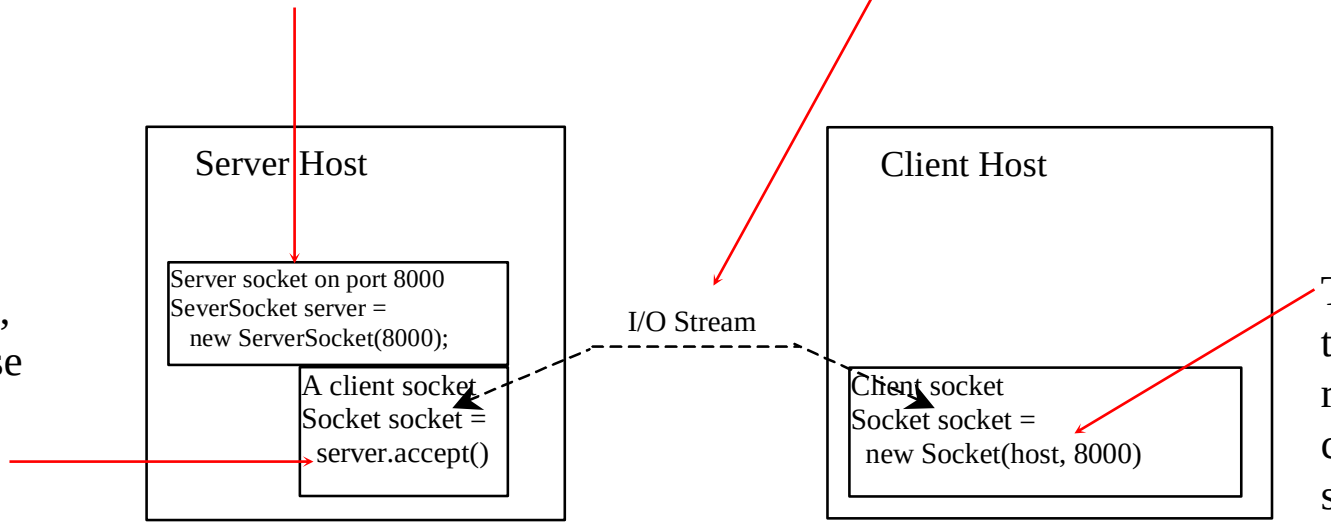
- On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server.

- The client and server can now communicate by writing to or reading from their sockets.

Client/Server Communications

The server must be running when a client starts. The server waits for a connection request from a client. To establish a server, you need to create a server socket and attach it to a port, which is where the server listens for connections.

After the server accepts the connection, communication between server and client is conducted the same as for I/O streams.



After a server socket is created, the server can use this statement to listen for connections.

The client issues this statement to request a connection to a server.

Step – 1: Creating a *ServerSocket*

- Establishing a simple server in Java requires five steps.

```
ServerSocket serverSocket = new ServerSocket(portNumber,
                                             queueLength)
```

- Registers an available TCP port number and specifies the maximum number of clients that can wait to connect to the server

Step – 2: Wait for a Connection

- Programs manage each client connection with a **Socket** object.
- Step 2 is now server listens for the connections indefinitely.
- To listen for a client connection, the program calls `ServerSocket` method **`accept`**

```
Socket socket = serverSocket.accept();
```

- This returns a `Socket` when a connection with a client established.

Step – 3: Get the Socket's I/O Streams

- *Step 3* is to get the OutputStream and InputStream objects that enable the server to communicate with the client by sending and receiving bytes.
- The server sends information to the client via an OutputStream and receives information from the client via an InputStream.
- The server invokes method **getOutputStream** on the Socket to get a reference to the Socket's OutputStream and invokes method **getInputStream** on the Socket to get a reference to the Socket's InputStream.
- Often it's useful to send or receive values of primitive types (e.g., int and double) or Serializable objects (e.g., Strings or other serializable types) rather than sending bytes.

```
DataInputStream inputFromClient = new DataInputStream(socket.getInputStream());
DataOutputStream outputToClient = new
                                    DataOutputStream(socket.getOutputStream());
```

Socket class

- A socket is simply an endpoint for communications between the machines.
- The Socket class can be used to create a socket.

public InputStream getInputStream()	returns the InputStream attached with this socket.
public OutputStream getOutputStream()	returns the OutputStream attached with this socket.
void close()	closes this socket

ServerSocket class

- The ServerSocket class can be used to create a server socket.
- This object is used to establish communication with the clients.

public Socket accept()	returns the socket and establish a connection between server and client.
void close()	closes the server socket.

The InetAddress Class

Occasionally, you would like to know who is connecting to the server. You can use the InetAddress class to find the client's host name and IP address. The InetAddress class models an IP address. You can use the statement shown below to create an instance of InetAddress for the client on a socket.

```
InetAddress inetAddress = socket.getInetAddress();
```

Next, you can display the client's host name and IP address, as follows:

```
System.out.println("Client's host name is " + inetAddress.getHostName());
System.out.println("Client's IP Address is " + inetAddress.getHostAddress());
```

Establishing a Simple Client
Step 1 – Create Socket to connect to server

- First step is create a socket to connect to the server.

```
Socket connection = new Socket(serverAddress, portNumber);
```

- If the connection attempt is successful it will return a Socket.

Step 2 – Get the Socket's I/O Stream

- The client uses Socket methods getInputStream and getOutputStream to obtain references to the Socket's InputStream and OutputStream.

- If the server is sending information in the form of actual types, the client should receive the information in the same format. Thus, if the server sends values with an ObjectOutputStream, the client should read those values with an ObjectInputStream.

Step 3 – Perform the Processing

Step 3 is the processing phase in which the client and the server communicate via the InputStream and OutputStream objects.

Step 3 – Perform the Processing

- In *Step 4*, the client closes the connection when the transmission is complete by invoking the close method on the streams and on the Socket.
- The client must determine when the server is finished sending information so that it can call close to close the Socket connection.