# Network Programming with J2ME Wireless Devices

*by Yu Feng*

## Overview

The wireless devices such as cell phones and two-way pagers keep their owners connected to the outside world at anytime from anywhere. They offer great connectivity that other types of devices couldn't offer. Application development for these wireless devices is going to be in great demand for the next couple years. Network programming plays an important role in wireless application development to take advantage of the connectivity these devices have to offer.

Sun's Java 2 Micro Edition (*J2ME*) offers a great development platform for developing applications for the embedded electronics and mobile devices. In Java 2 Micro Edition (*J2ME* ), the Connected Limited Device Configuration (*CLDC*) defines a generic "configuration" for a broad range of handheld devices. On top of CLDC, the Mobile Information Device Profile (*MIDP*) is defined specifically for wireless devices such as cell phones and two-way pagers. Wireless device manufacturers need to implement MIDP in order to support Java applications on their devices. For example, Motorola is going to release the MIDP implementation for its iDEN mobile phone family in the Q1 of 2001. Research In Motion is also going to release the MIDP implementation for its Blackberry pager family soon.

In the next section, we will take a look what's new in J2ME network programming. Later in this article, we will examine the HttpConnection class in J2ME MIDP. We will conclude with a complete example of how to use the HttpConnection class.

## The J2ME Generic Connection Framework

In Java 2 Standard Edition (*J2SE*), network programming is fairly straightforward. Most of the network related classes are packaged in java.net. There are about 20 classes available in this package. These classes provide a rich set of functionality to support the network communication in Java applications. However, the java.net package is not suitable for the wireless applications that run on cell phones and two-way pagers. The size of java.net is around 200KB. It is too big to fit in with the wireless devices that only have a few hundred kilobytes of total memory and storage budget. Besides the size issue, there is another challenge when dealing with wireless devices: J2ME needs to support a variety of mobile devices that come with different sizes and shapes, different networking capabilities, and different file I/O requirements.

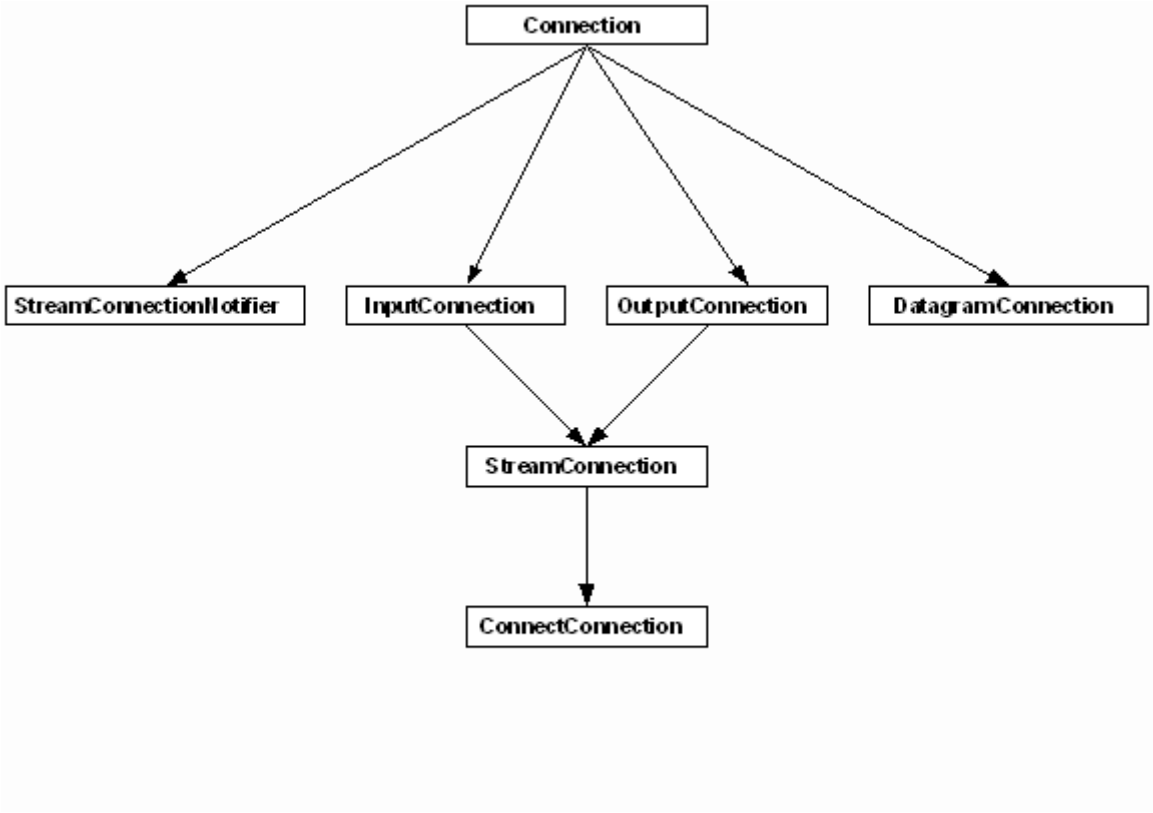
The networking in J2ME has to be very flexible to support a variety of devices and has to be very device specific at the same time. To meet these challenges, the Generic Connection framework is first introduced in the CLDC. The idea of the Generic Connection framework is to define the abstractions of the networking and file I/O as general as possible to support a broad range of handheld devices, and leave the actual implementations of these abstractions to individual device manufacturers. These abstractions are defined as Java interfaces. The device manufacturers choose which one to implement in their MIDP based on the actual device capabilities.

There is 1 class (Connector) and 7 connection interfaces (Connection, ContentConnection, DatagramConnection, InputConnection, OutputConnection, StreamConnection, and StreamConnectionNotifier) defined in the Generic

Connection framework. They can be found in the javax.microedition.io package that comes with J2ME CLDC. Please note that there is no implementation of the connection interfaces at the CLDC level. The actual implementation is left to MIDP.

The 7 connection interfaces define the abstractions of 6 basic types of communications: basic serial input, basic serial output, datagrams communications, sockets communications, notification mechanism in a client-server communication, and basic HTTP communication with a Web server.

The relationships between these interfaces are illustrated in the following diagram:




As shown in the figure, Connection is the base interface and the root of the connection interface hierarchy. All the other connection interfaces derive from Connection. StreamConnection derives from InputConnection and OutputConnection. It defines both the input and output capabilities for a stream connection. ContentConnection derives from StreamConnection. It adds three additional methods from MIME handling on top of the I/O methods in StreamConnection.

The Connector class is the core of Generic Connection framework. The reason we say that is because all connection objects we mentioned above are created by the static method open defined in Connector class. Different types of communication can be created by the same method with different parameters. The connection could be file I/O, serial port communication, datagram connection, or an http connection depending on the string parameter passed to the method. Such design makes J2ME implementation very extensible and flexible in supporting new devices and products.

Here is how the method is being used:

```
Connector.open(String connect);
```

The parameter connect is a String variable. It has a URL-like format: {protocol}:[{target}][{params}] and it consists of three parts: protocol, target, and params.

protocol dictates what type of connection will be created by the method. There are several possible values for protocol: file, socket, comm, datagram and http. file indicates that the connection will be used for file I/O, comm indicates that the connection will be used for serial port communication, and http indicates that the connection is created for accessing web servers.

target can be a host name, a network port number, a file name, or a communication port number.

params is optional, it specifies the additional information needed to complete the connect string.

The following examples illustrate how to use the open method to create different types of communication based on different protocols:

*HTTP communication:*

```
Connection hc = Connector.open("http://www.wirelessdevnet.com");
```

*Stream-based Socket communication:*

```
Connection sc = Connector.open("socket://localhost:9000");
```

*Datagram-based socket communication:*

```
Connection dc = Connector.open("datagram://:9000");
```

*Serial port communication:*

```
Connection cc = Connector.open("comm:0;baudrate=9000");
```

*File I/O*

```
Connection fc = Connector.open("file://foo.dat");
```

As we mentioned earlier, the support for these protocols varies from vendor to vendor. Developers have to check the documentation from each MIDP device manufacturers to see if they support the specific protocol. I tested the socket, datagrams, and http connections with MotoSDK (a development kit from Motorola).

All three types of connections are supported by the Motorola's MIDP. Sun's MIDP reference implementation supports only the http connections. We will talk more about this later. If your program is trying to create a connection based on a protocol that is not supported by your device manufacturer, a ConnectionNotFoundException exception will be thrown.

## The HttpConnection Class

The HttpConnection class is defined in J2ME MIDP to allow developer to handle http connections in their wireless applications. The HttpConnection class is guaranteed available on all MIDP devices.

Theoretically you may use either sockets, or datagrams for remote communication in your J2ME applications if your MIDP device manufacturer supports them. But this creates a portability issue for your applications, because you can't always count on that other device manufacturers will support them as well. This means that your program may run on one MIDP device but fail on others. So you should consider using the HttpConnection class in your application first, because HTTPConnection is mandatory for all MIDP implementations.

In this section, we will take a hard look at the HttpConnection class. A complete sample program using HttpConnection is shown in the next section.

The HttpConnection interface derives from the ContentConnection interface in CLDC. It inherits all I/O stream methods from StreamConnection, all the MIME handling methods from ContentConnection and adds several additional methods for handling http protocol specific needs.

Here is a laundry list of all the methods available in HttpConnection:

**I/O related:**

DataInputStream openDataInputStream()

InputStream openInputStream()

DataOutputStream openDataOutputStream()

OutputStream openOutputStream()

An input stream can be used to read contents from a Web server while output stream can be used to send request to a Web server. These streams can be obtained from the HttpConnection object after the connection has been established with the Web Server. In the following example, an http connection is established with Web server www.wirelessdevnet.com at port 80 (the default http port). Then an input stream is obtained for reading response from the Web:

```
HttpConnection hc = (HttpConnection);
Connector.open("http://www.wirelessdevnet.com");
InputStream is = new hc.openInputStream();

      int ch;
      // Check the Content-Length first
      long len = hc.getLength();
```

```
      if(len!=-1) {
        for(int i = 0;i<len;i++)
         if((ch = is.read())!= -1)
          System.out.print((char) ch));
      } else {
        // if the content-length is not available
        while ((ch = is.read()) != -1)
          System.out.print((char) ch));
      }

is.close();
hc.close();
```

**MIME related:**

String getEncoding()

long getLength()

String getType()

Once the http connection is established, these three methods can be used to obtain the value of the following three fields in the HTTP header: Content-Length, Content-Encoding and Content-Type. For more information on HTTP protocol, please visit http://www.w3.org/Protocols/rfc2068/rfc2068.

**Http Protocol related:**

long getDate()

long getExpiration()

String getFile()

String getHeaderField(int index)

String getHeaderField(String name)

long getHeaderFieldDate(String name, long def)

int getHeaderFieldInt(String name, int def)

String getHeaderFieldKey(int n)

String getHost()

long getLastModified()

int getPort()

String getProtocol()

String getQuery()

String getRef()

String getRequestMethod()

String getRequestProperty(String key)

int getResponseCode()

String getResponseMessage()

String getURL()

void setRequestMethod(String method)

void setRequestProperty(String key, String value)

GetResponseCode and getResponseMessage can be used to check the response status code and message from the Web server. In this typical response message from a Web server: "HTTP/1.1 401 Unauthorized", you will get a response code "401" and a response message "Unauthorized".

The following methods are used for obtaining additional header information: getDate, getExpiration, getFile, getHeaderField, getHeaderField, getHeaderFieldDate, getHeaderFieldInt, getHeaderFieldKey and getLastModified

The following methods are used for retrieving the individual components parsed out of the URL string: GetHost, getPort, getProtocol, getQuery, getRef, getRequestMethod, getRequestProperty, getResponseCode, getResponseMessage, and getURL.

These three methods are used for dealing with the HEAD, GET, and POST requests: setRequestMethod, setRequestProperty, and getRequestMethod.

## A Complete Example

The following example is a MIDlet program that communicates with a Web server to perform a string reversing operation.

Here is the program flow:

1. User types in a string on the phone (see Figure 2)


*Figure 2: Entering The String*

2. The string is then sent to a Java servlet on a Web server using the HTTP POST method. The servlet is located at http://www.webyu.com/servlets/webyu/wirelessdevnetsample1.
3. After the Java servlet receives the string, it reverses the string and sends it back to the MIDlet application for display (see Figure 3).


*Figure 3: Displaying the reversed string*

This is a trivial string operation that can be dealt with locally on the phone. However, the purpose of this program is to demonstrate how to use HttpConnection in a MIDlet program.

## Code Listing: Sample1.java

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.io.*;
import java.io.*;

public class Sample1 extends MIDlet
        implements CommandListener {
/*
* the default value for the URL string is
* http://www.webyu.com/servlets/webyu/wirelessdevnetsample1
*/

private static String defaultURL =
"http://www.webyu.com/servlets/webyu/wirelessdevnetsample1";
```

```
// GUI component for user to enter String
private Display myDisplay = null;
private Form mainScreen;
private TextField requestField;

// GUI component for displaying header information
private Form resultScreen;
private StringItem resultField;

// the "SEND" button used on the mainScreen
Command sendCommand = new Command("SEND", Command.OK, 1);

// the "BACK" button used on the resultScreen
Command backCommand = new Command("BACK", Command.OK, 1);

public Sample1(){
        // initializing the GUI components for entering Web URL
        myDisplay = Display.getDisplay(this);
        mainScreen = new Form("Type in a string:");
        requestField =
                new TextField(null, "GREAT ARTICLE", 100,
TextField.ANY);
        mainScreen.append(requestField);
        mainScreen.addCommand(sendCommand);
        mainScreen.setCommandListener(this);
}

public void startApp() {
        myDisplay.setCurrent(mainScreen);
}

public void pauseApp() {
}

public void destroyApp(boolean unconditional) {
}

public void commandAction(Command c, Displayable s) {
        if (c == sendCommand) {
                // retrieving the String that user entered
                String requeststring = requestField.getString();
                // sending a POST request to Web server
                String resultstring = sendPostRequest(requeststring);
                // displaying the response back from Web server
                resultScreen = new Form("Result String:");
                resultField = new StringItem(null, resultstring);
                resultScreen.append(resultField);
                resultScreen.addCommand(backCommand);
                resultScreen.setCommandListener(this);
                myDisplay.setCurrent(resultScreen);
        } else if (c == backCommand) {
                // do it all over again
                requestField.setString("SOMETHING GOOD");
                myDisplay.setCurrent(mainScreen);
        }
}

// send a POST request to Web server
public String sendPostRequest(String requeststring) {
```

```
HttpConnection hc = null;
DataInputStream dis = null;
DataOutputStream dos = null;
StringBuffer messagebuffer = new StringBuffer();
try {
        // Open up a http connection with the Web server
        // for both send and receive operations
        hc = (HttpConnection)
        Connector.open(defaultURL, Connector.READ_WRITE);
        // Set the request method to POST
        hc.setRequestMethod(HttpConnection.POST);
        // Send the string entered by user byte by byte
        dos = hc.openDataOutputStream();
        byte[] request_body = requeststring.getBytes();
        for (int i = 0; i < request_body.length; i++) {
                dos.writeByte(request_body[i]);
        }
        dos.flush();
        dos.close();
        // Retrieve the response back from the servlet
        dis = new DataInputStream(hc.openInputStream());
        int ch;
        // Check the Content-Length first
        long len = hc.getLength();
        if(len!=-1) {
                for(int i = 0;i
```