

A constant for the `LEFT` game action

`RIGHT`

A constant for the `RIGHT` game action

`FIRE`

A constant for the `FIRE` game action

`GAME_A`

A constant for the general purpose "A" game action

`GAME_B`

A constant for the general purpose "B" game action

`GAME_C`

A constant for the general purpose "C" game action

`GAME_D`

A constant for the general purpose "D" game action

While each keycode is mapped to one game action, a game action can be associated with more than one keycode. The translation between the two is done with the `getKeyCode()` and `getGameAction()` methods.



If your application uses game actions and you want the application to be portable, you should translate key events into game actions with the `getGameAction()` method and test the result. For example, the game actions `UP`, `DOWN`, `LEFT`, and `RIGHT` can be mapped differently on different devices. The `getGameAction()` method returns the `RIGHT` game action, for example, when the user presses the key that is a natural right on the device.

6.2.3 Event Delivery Methods

The following methods of the `Canvas` class are available for handling low-level events:

```
protected void keyPressed(int keyCode);
protected void keyReleased(int keyCode);
protected void keyRepeated(int keyCode);
protected void pointerPressed(int x, int y);
protected void pointerDragged(int x, int y);
protected void pointerReleased(int x, int y);
```

These methods are callbacks that you should override in a class that extends the `Canvas` class. There are a couple of important things to note here:

- The `keyRepeated()` method might not be available on all devices. Your application should check the availability of repeat actions by calling the `hasRepeatEvents()` method.
- The pointer events may not be present on all devices, so before using the `pointerPressed()`, `pointerDragged()`, and `pointerReleased()` methods, your application should check if a pointer mechanism is available by calling the `hasPointerEvents()` and `hasPointerMotionEvents()` methods first.

6.2.3.1 Handling low-level events

[Example 6-4](#) shows how to handle low-level events. In this example, the `Canvas` class is subclassed to use an anonymous inner class, an implementation is provided for the `keyPressed()` and `keyReleased()` methods, and an empty implementation is provided for the `paint()` method. When a key is pressed or released, the application prints the value of that key.

Example 6-4. Handling low-level events

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

public class EventEx4 extends MIDlet {
    Display display;
    Command exit;

    public EventEx4() {
        display = Display.getDisplay(this);
    }

    public void destroyApp (boolean unconditional) {
    }

    public void pauseApp () {
        System.out.println("App paused.");
    }

    public void startApp () {
        display = Display.getDisplay(this);

        Canvas canvas = new Canvas() { // anonymous class
            public void paint(Graphics g) {
            }

            protected void keyPressed(int keyCode) {
                if (keyCode > 0) {
                    System.out.println("keyPressed " + ((char) keyCode));
                } else {
                    System.out.println("keyPressed action " +
                        +getGameAction(keyCode));
                }
            }

            protected void keyReleased(int keyCode) {
                if (keyCode > 0) {

```

```

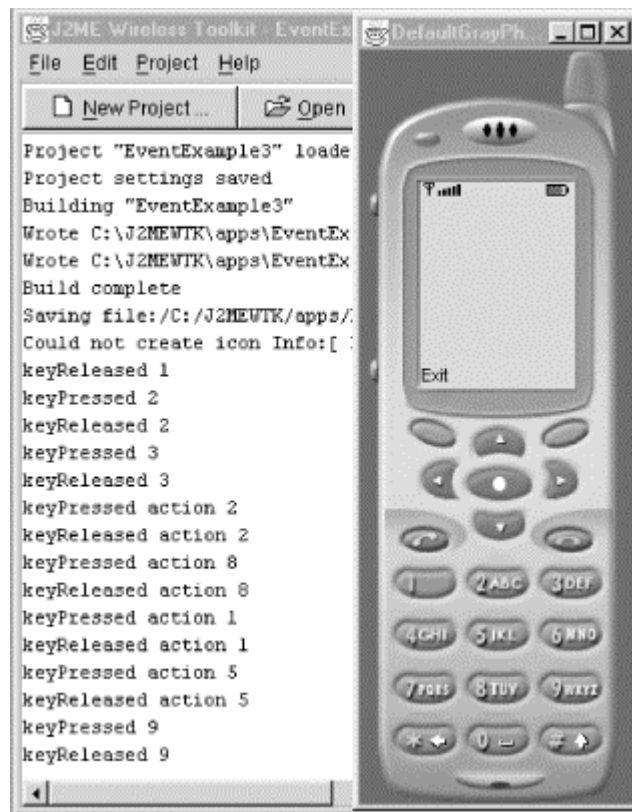
        System.out.println("keyReleased " + ((char)keyCode));
    } else {
        System.out.println("keyReleased action "
            + getGameAction(keyCode));
    }
}
}; // end of anonymous class

exit = new Command("Exit", Command.STOP, 1);
canvas.addCommand(exit);
canvas.setCommandListener(new CommandListener() {
    public void commandAction(Command c, Displayable d) {
        if(c == exit) {
            notifyDestroyed();
        } else {
            System.out.println("Saw the command: "+c);
        }
    }
});
display.setCurrent(canvas);
}
}

```

If you run the `EventEx3` MIDlet and activate it, you should see output similar to that in [Figure 6-2](#).

Figure 6-8. Handling low-level events



An alternative implementation for the `keyPressed()` method is to interpret the keys at runtime, as shown in the following segment of code:

```

public void keyPressed(int keyCode) {
    int action = getGameAction(keyCode);
    switch(action) {
        case LEFT: System.out.println("MOVE TO THE LEFT");break;
        case RIGHT: System.out.println("MOVE TO THE RIGHT");break;
        // and so on....
    }
}

```

Chapter 7. Networking

Way back in [Chapter 1](#), we briefly introduced the CLDC Generic Connection Framework. Let's quickly review why it was necessary to create an entirely new networking library for the CLDC.

The `java.io` and `java.net` packages of the J2SE are not suitable for handheld devices with a small memory footprint, for the following reasons:

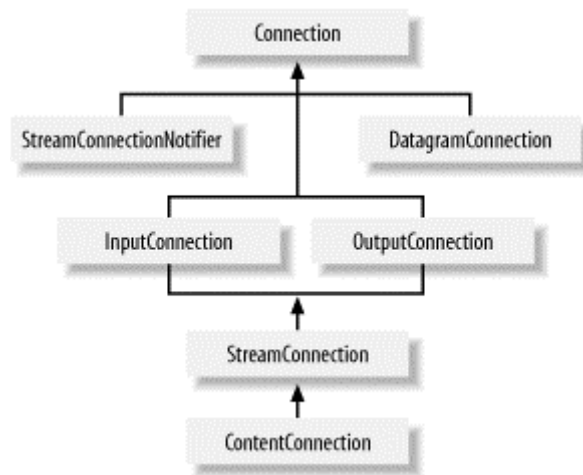
- Device manufacturers who work with circuit-switched networks require stream-based connections such as the Transport Control Protocol (TCP), which is a connection-oriented protocol.
- Device manufacturers working with packet-switched networks require datagram-based connections such as the User Datagram Protocol (UDP), which is a connectionless protocol.
- Other handheld devices have specific mechanisms for communications.

All this variation makes designing networking facilities for the CLDC quite a challenge. This challenge has led to the design of a set of related abstractions that can be used at the programming level instead of using different abstractions for different forms of communications. For example, the J2SE `java.net` package provides a set of related abstractions in the form of over 20 networking classes, including `Socket`, `ServerSocket`, and `DatagramSocket`. With the CLDC, however, we need to go a step further to save space.

7.1 Generic Connections

In the Generic Connection Framework, all connections are created using the static `open( )` methods from a single class: `javax.microedition.io.Connector`. If successful, these methods return an object that implements one of the generic connection interfaces. [Figure 7-1](#) shows how these interfaces form an inheritance hierarchy. The `Connection` interface (don't confuse `Connection` with `Connector`) is the base interface.

Figure 7-1. Connection interface hierarchy



- The `Connection` interface represents the most basic connection type. It can only be opened and closed.
- The `InputConnection` interface represents a device from which data can be read. Its `openInputStream()` method returns an input stream for the connection.
- The `OutputConnection` interface represents a device to which data can be written. Likewise, its `openOutputStream()` method returns an output stream for the connection.
- The `StreamConnection` interface combines the input and output connections.
- The `ContentConnection` interface extends the `StreamConnection` interface. It provides access to some of the basic metadata information provided by HTTP connections.
- The `StreamConnectionNotifier` interface waits for a connection to be established. It returns an object that implements the `StreamConnection` interface, on which a communication link has been established.
- The `DatagramConnection` interface is used to represent a datagram endpoint.

The simplest `open()` method of the `Connector` class has the following syntax:

```
public Connection open(String name) throws java.io.IOException;
```

The `String` parameter has the format "`scheme:targetaddress;parameters`" and conforms to the URL syntax in RFC 2396. Here are a few examples:

Establish an HTTP connection

```
Connector.open("http://www.ora.com");
```

Start a socket connection

```
Connector.open("socket://www.ora.com:80");
```

Establish a datagram connection

```
Connector.open("datagram://192.168.2.101:2345");
```

Communicate with a port

```
Connector.open("comm:0;baudrate=9600");
```

Open files

```
Connector.open("file:/myfile.txt");
```

The goal of the above syntax is to isolate the differences between the setup of one protocol and another protocol into a simple string that characterizes the type of connection. Most of the application's code remains the same, regardless of the protocol you use. You will see the benefits of this in the examples later in this chapter.



The connection examples above are for illustration only. The CLDC itself does not provide any protocol implementations, because no implementations should be provided at the configuration level. In addition, a J2ME profile such as the MIDP does not have to provide implementations for all of the protocols mentioned earlier. The MIDP implementation from Sun Microsystems, for example, provides an implementation for the HTTP protocol, but it does not provide implementations for socket or datagram connections.

The `Connector` class provides two other static `open( )` methods that can be used to open connections, which have the following signatures:

```
public static Connection open(String url, int mode)
    throws java.io.IOException;
public static Connection open(String url, int mode, boolean timeouts)
    throws java.io.IOException;
```

The first method takes two parameters: the URL for the connection and the access mode. The access mode is used to indicate to the protocol handler the intentions of the calling code. The access mode can be one of the following constants, which are defined in the `Connector` class:

```
public static final int READ;
public static final int WRITE;
public static final int READ_WRITE;
```

You can use these constants to specify if the connection is going to be exclusively read from (`READ`), exclusively written to (`WRITE`), or both (`READ_WRITE`). If the access mode parameter is omitted, the `READ_WRITE` default will be used. It is important, however, to note that these flags are protocol-dependent. For example, a connection to a printer would not allow `READ` access.

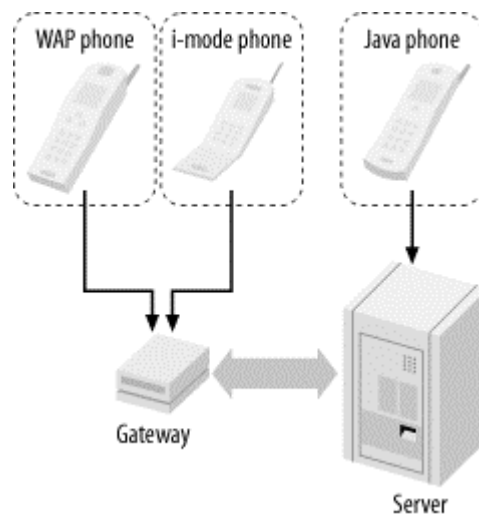
The other `open( )` method takes an additional third parameter, which is a boolean flag to indicate that the calling code wants to receive a timeout exception in the form of a `java.io.InterruptedIOException`. The timeout value is not given, as it is protocol-dependent, and there is no guarantee that the underlying protocol

implementation will throw the timeout exception. If this parameter is omitted, then no timeout exceptions will be thrown.

7.2 MIDP Connectivity

The MIDP extends the CLDC Generic Connection Framework to provide support for the HTTP protocol. Why HTTP? Well, HTTP can be implemented using both IP protocols (such as TCP/IP) or non-IP protocols (such as WAP and I-mode). In the latter case, the device would have to utilize a gateway that could perform URL naming resolution to access the Internet, as shown in [Figure 7-2](#).

Figure 7-2. The benefit of HTTP support



All of the MIDP 1.0 implementations must provide support for the HTTP protocol. Therefore, we encourage you to only use protocols supported by the MIDP (i.e., HTTP), as this will allow the application to be portable across all mobile information devices.

The idea of having the MIDP support the HTTP protocol is very clever. For network programming, you can revert to the HTTP programming model, and your applications will run on any MIDP device, whether it is a GSM phone with a WAP stack, a phone with I-mode, a Palm VII wireless, or a handheld device with Bluetooth.

7.2.1 The `HttpConnection` Interface

The `HttpConnection` interface is part of the `javax.microedition.io` package. This interface defines the necessary methods and constants to exchange data through an HTTP connection. It has the following methods (the constants, which have been omitted here to save space, are documented in [Appendix D](#)):

```
public interface HttpConnection extends ContentConnection {
    // public instance methods
    public long getDate( ) throws IOException;
    public long getExpiration( ) throws IOException;
    public String getFile( );
}
```

```

    public String getHeaderField(int n) throws IOException;
    public String getHeaderField(String name) throws IOException;
    public long getHeaderFieldDate(String name, long def) throws
IOException;
    public int getHeaderFieldInt(String name, int def) throws
IOException;
    public String getHeaderFieldKey(int n) throws IOException;
    public String getHost( );
    public long getLastModified( ) throws IOException;
    public int getPort( );
    public String getProtocol( );
    public String getQuery( );
    public String getRef( );
    public String getRequestMethod( );
    public String getRequestProperty(String key);
    public int getResponseCode( ) throws IOException;
    public String getResponseMessage( ) throws IOException;
    public String getURL( );
    public void setRequestMethod(String method) throws IOException;
    public void setRequestProperty(String key, String value) throws
IOException;
}

```

The HTTP protocol is a request-response application protocol in which the parameters of the request must be set before the request is sent. The connection could be in one of the three following states:

Setup

No connection has been established yet.

Connected

A connection has been made and the request has been sent; a response is expected soon.

Closed

The connection has been closed. Some methods will throw an `IOException` if called.

Only in the setup state can the methods `setRequestMethod( )` and `setRequestProperty( )` be invoked. These can be used to cover the HTTP headers that are typically seen in an HTTP request. For example, suppose you have the following connection:

```

HttpConnection c = (HttpConnection)
    Connector.open("http://www.ora.com");

```

You can set the request method to be of type `POST` as follows:

```

c.setRequestMethod(HttpConnection.POST);

```

You can also use the `setRequestProperty( )` method to set some of the HTTP header information. For example, you can set the `User-Agent` property as follows:

```
c.setRequestProperty("User-Agent",
    "Profile/MIDP-1.0 Configuration/CLDC-1.0");
```

The following methods of `HttpConnection` (or its sub-interfaces), which cause data to be transmitted or received, will cause a state transition from the setup state to the connected state:

```
public long getDate( ) throws java.io.IOException
public String getEncoding( )
public long getExpiration( ) throws java.io.IOException
public String getHeaderField(String name) throws java.io.IOException
public long getHeaderFieldDate(String name, long def) throws
java.io.IOException
public int getHeaderFieldInt(String name, int def) throws
java.io.IOException
public String getHeaderFieldKey(int n) throws java.io.IOException
public long getLastModified( ) throws java.io.IOException
public long getLength( )
public int getResponseCode( ) throws java.io.IOException
public String getResponseMessage( ) throws java.io.IOException
public String getType( )
public DataInputStream openDataInputStream(String name) throws
java.io.IOException
public DataOutputStream openDataOutputStream(String name) throws
java.io.IOException
public InputStream openInputStream(String name) throws
java.io.IOException
public OutputStream openOutputStream(String name) throws
java.io.IOException
```

While the connection is open (i.e., in the connected state), the following methods can be safely invoked:

```
public void close( ) throws java.io.IOException
public String getFile( )
public String getHost( )
public int getPort( )
public String getProtocol( )
public String getQuery( )
public String getRequestMethod( )
public String getRequestProperty(String key)
public String getURL( )
```

Before we look at sample applications, let's briefly review some of the concepts that our examples will be using: the HTTP programming model, CGI, and Java servlets.

7.3 The HTTP Programming Model

HTTP is a request-response application protocol. When programming with Java HTTP libraries, such as the Generic Connection Framework, the parameters of the request must always be set before the request is sent. This allows the entire request, including parameters, to be sent at the same time.

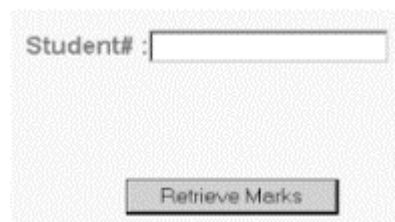
7.3.1 Request Methods

There are two commands to send data from a form on a web page to a CGI script or a servlet hosted by the HTTP server. These commands are `GET` and `POST`. Each of these has a different way of sending data to the server.

- For the `GET` method, the input values are sent as part of the URL in the `QUERY_STRING` environment variable.
- For the `POST` method, data is sent as an input stream and its length is saved in the `CONTENT_LENGTH` environment variable.

The `POST` method is more secure, and you can send more data using it. As an example, consider the following HTML code for the form shown in [Figure 7-3](#).

Figure 7-3. Form with one field (GET method)



```
<form action=http://www.somesite.com/cgi-bin/getgrade.cgi-  
method="GET">  
Student#:  
<input type="text" name="idnum" size=30>  
<input name="RetrieveMarks" value="Retrieve Marks" type="submit">  
</form>
```

This form is handled by the script at *http://www.somesite.com/cgi-bin/getgrade.cgi*. Note that the form uses the `GET` method to transmit the information. When the user enters a student number, such as 112233, and clicks the Retrieve Marks button, the form data is sent to the CGI script as part of the URL. Hence, the encoded URL is: *http://www.somesite.com/cgi-bin/getgrade.cgi?idnum=112233*.

In the case of `POST`, however, input values are not sent as part of the URL. They are sent as an input stream in a separate message.

If the user enters a string with spaces, all spaces are replaced by the pluses (+). Also, if the form requires multiple input values for different fields, the fields are separated by an ampersand (&). For example, if the above form has two input fields—one for the student name and the other for the student number—the names of the fields are `name` and `idnum`, respectively. Suppose the input values are "Sam Lee" for `name` and "112233" for `idnum`. Then the encoded URL would be: *http://www.somesite.com/cgi-bin/getgrade.cgi?name=Sam+Lee&idnum=112233*.

7.3.2 Servlets

Java servlets also support a request and response programming model. When a client sends a request to the server, the server relays the request to the servlet. The servlet then constructs a response that the server relays back to the client. Unlike CGI scripts,

however, servlets are written in Java and run within the same process as the HTTP server.

When a client request is made, the server first calls upon the `service()` method of the servlet and passes it a request and response object. The servlet then determines whether this request is a GET or POST operation, and calls either the `HttpServlet.doGet()` or `HttpServlet.doPost()` methods as needed. Both the `doGet()` and `doPost()` methods take a request object, `HttpServletRequest`, and a response object, `HttpServletResponse`, as parameters.^[1]

^[1] This is just enough information to get you through this chapter. If you'd like to learn more about Java servlets, we recommend *Java Servlet Programming* by Jason Hunter (O'Reilly).

7.4 Invoking Remote Applications from MIDlets

Now let's look at some examples of fetching HTTP pages and invoking CGI scripts and servlets from MIDlets using the connection framework.

7.4.1 Fetching a Page

[Example 7-1](#) shows how to read the contents of a file referenced by a URL, using a `StreamConnection`. An `HttpConnection` can also be used, but since no HTTP-specific behavior is needed here, the `StreamConnection` is used. The application is very simple. The `Connector.open()` method opens a connection to the URL and returns a `StreamConnection` object. Then an `InputStream` is opened through which to read the contents of the file, one character at a time, until the end of the file (signaled by a character value of -1) is reached. In the event that an exception is thrown, both the stream and connection are closed.

Example 7-1. Fetching a page referenced by a URL

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

public class FetchPageMidlet extends MIDlet {

    private Display display;

    String url = "http://www.javacourses.com/hello.txt";

    public FetchPageMidlet() {
        display = Display.getDisplay(this);
    }

    /**
     * This will be invoked when we start the MIDlet
     */
    public void startApp() {
        try {
            getViaStreamConnection(url);
        } catch (IOException e) {
            //Handle Exceptions any other way you like.
            System.out.println("IOException " + e);
        }
    }
}
```

```

        e.printStackTrace( );
    }
}

/**
 * Pause, discontinue ....
 */
public void pauseApp( ) {

}

/**
 * Destroy must cleanup everything.
 */
public void destroyApp(boolean unconditional) {

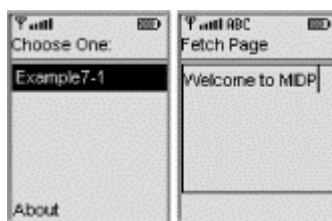
}

/**
 * read url via stream connection
 */
void getViaStreamConnection(String url) throws IOException {
    StreamConnection c = null;
    InputStream s = null;
    StringBuffer b = new StringBuffer( );
    TextBox t = null;
    try {
        c = (StreamConnection)Connector.open(url);
        s = c.openInputStream( );
        int ch;
        while((ch = s.read( )) != -1) {
            b.append((char) ch);
        }
        System.out.println(b.toString( ));
        t = new TextBox("Fetch Page", b.toString( ), 1024, 0);
    } finally {
        if(s != null) {
            s.close( );
        }
        if(c != null) {
            c.close( );
        }
    }
    // display the contents of the file in a text box.
    display.setCurrent(t);
}
}

```

When you run and activate `FetchPageMidlet`, you should see a screen similar to [Figure 7-4](#).

Figure 7-4. Fetching a page reference by a URL



7.4.2 Invoking a CGI Script (GET)

The following example shows how to invoke a CGI script from an HTTP server, capturing and displaying the results on the handheld device screen. This example uses an `HttpConnection` object, which is returned by the call to the `Connector.open( )` method. The call to `setRequestMethod( )` sets the request method to `GET`, followed by two calls to `setRequestProperty( )`, which in turn set the HTTP request header properties `User-Agent` and `Content-Language`. In this example, a script is invoked with a student ID number encoded as part of the URL. The script searches a database file and returns the final grade for the student corresponding to the ID number.



The URL has the form *http://www.javacourses.com/cgi-bin/getgrade?idnum=182016*. Alternatively, you can do this with the commented-out code shown below in bold. The latter technique can also be used if the student ID number is to be entered by the user, as you will see in the next example.

This example's source code is shown in [Example 7-2](#).

Example 7-2. Invoking a CGI script (GET method)

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

/**
 * An example MIDlet to invoke a CGI script (GET method).
 */

public class InvokeCgiMidlet1 extends MIDlet {

    private Display display;

    String url =
"http://www.javacourses.com/cgi-bin/getgrade.cgi?idnum=182016";

    public InvokeCgiMidlet1( ) {
        display = Display.getDisplay(this);
    }

    /**
     * Initialization. Invoked when we activate the MIDlet.
     */
    public void startApp( ) {
        try {
            getGrade(url);
        } catch (IOException e) {
            System.out.println("IOException " + e);
            e.printStackTrace( );
        }
    }

    /**
     * Pause, discontinue ....
     */
}
```

```

    */
    public void pauseApp( ) {
    }

    /**
     * Destroy must cleanup everything.
     */
    public void destroyApp(boolean unconditional) {
    }

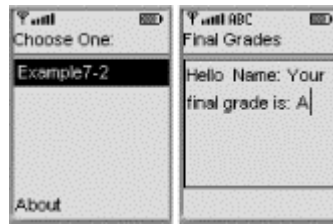
    /**
     * Retrieve a grade....
     */

    void getGrade(String url) throws IOException {
        HttpURLConnection c = null;
        InputStream is = null;
        OutputStream os = null;
        StringBuffer b = new StringBuffer( );
        TextBox t = null;
        try {
            c = (HttpURLConnection)Connector.open(url);
            // set the request method to GET
            c.setRequestMethod(HttpURLConnection.GET);
            // set some HTTP request headers
            c.setRequestProperty("User-Agent","Profile/MIDP-1.0
Configuration/CLDC-1.0");
            c.setRequestProperty("Content-Language", "en-CA");
            os = c.openOutputStream( );
            /*
            //Retrieve info for ID number 182016
            String str = "?idnum=182016";
            byte postmsg[] = str.getBytes( );
            for(int i=0;i<postmsg.length;i++) {
                os.writeByte(postmsg[i]);
            }
            os.flush( );
            */
            is = c.openDataInputStream( );
            int ch;
            while ((ch = is.read( )) != -1) {
                b.append((char) ch);
                System.out.println((char)ch);
            }
            t = new TextBox("Final Grades", b.toString( ), 1024, 0);
        } finally {
            if(is!= null) {
                is.close( );
            }
            if(os != null) {
                os.close( );
            }
            if(c != null) {
                c.close( );
            }
        }
        display.setCurrent(t);
    }
}

```

When you run and activate `InvokeCgiMidlet1`, you should see a screen similar to [Figure 7-5](#).

Figure 7-5. Invoking a CGI script (GET method)



7.4.3 Invoking a CGI Script (POST)

Now let's talk about making HTTP requests using `POST`. In this example, the input is sent to the CGI script, called `pgrade.cgi`, in a message.

The URL variable, defined in [Example 7-3](#), specifies the location of the `pgrade.cgi` CGI script. In this example, an HTTP connection is opened to the CGI script, followed by opening input and output streams on the connection. Data for the script is sent through the output stream, and the response is received through the input stream. The CGI script in this example, which is written in Perl, takes a student number input value. If the student number is found in the database file, the script retrieves the corresponding final grade and returns the grade to the calling client. In the case of MIDlets, however, there are no HTML forms, so here the message `name=182016` is sent to the CGI script. The source code for this example is shown in [Example 7-3](#). Note that when using the `POST` method, the `CONTENT_TYPE` header must be set to `application/x-www-form-urlencoded`, because this content type:

- Specifies normal data encoding.
- Converts blanks to plus (+) signs.
- Converts non-alphanumeric characters to hexadecimal numbers preceded by a percent sign (%).
- Places an ampersand (&) between each `name=value` pair.

In short, this content type prevents data corruption during the transmission of form data from the browser to the server.

Example 7-3. Invoking a CGI script (POST method)

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

/**
 * An example MIDlet to invoke a CGI script (POST method is used).
 */

public class InvokeCgiMidlet2 extends MIDlet {

    private Display display;
```

```

String url = "http://www.javacourses.com/cgi-bin/pgrade.cgi";

public InvokeCgiMidlet2( ) {
    display = Display.getDisplay(this);
}

/**
 * Initialization. Invoked when we activate the MIDlet.
 */
public void startApp( ) {
    try {
        getGrade(url);
    } catch (IOException e) {
        System.out.println("IOException " + e);
        e.printStackTrace( );
    }
}

/**
 * Pause, discontinue ....
 */
public void pauseApp( ) {
}

/**
 * Destroy must cleanup everything.
 */
public void destroyApp(boolean unconditional) {
}

/**
 * Retrieve a grade....
 */

void getGrade(String url) throws IOException {
    HttpURLConnection c = null;
    InputStream is = null;
    OutputStream os = null;
    StringBuffer b = new StringBuffer( );
    TextBox t = null;
    try {
        c = (HttpURLConnection)Connector.open(url);
        c.setRequestMethod(HttpURLConnection.POST);
        c.setRequestProperty("CONTENT-TYPE",
            "application/x-www-form-urlencoded");
        c.setRequestProperty("User-Agent",
            "Profile/MIDP-1.0 Configuration/CLDC-1.0");
        c.setRequestProperty("Content-Language", "en-CA");
        os = c.openOutputStream( );

        // send input
        String str = "name=182016";
        byte postmsg[] = str.getBytes( );
        for(int i=0;i<postmsg.length;i++) {
            os.write(postmsg[i]);
        }
        os.flush( );
        is = c.openDataInputStream( );
        int ch;
        // receive output
        while ((ch = is.read( )) != -1) {

```

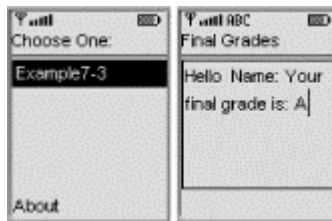
```

        b.append((char) ch);
        System.out.println((char) ch);
    }
    t = new TextBox("Final Grades", b.toString(), 1024, 0);
} finally {
    if(is != null) {
        is.close();
    }
    if(os != null) {
        os.close();
    }
    if(c != null) {
        c.close();
    }
}
display.setCurrent(t);
}
}

```

When you run and activate the `InvokeCgiMidlet2`, you should see output similar to [Figure 7-6](#).

Figure 7-6. Invoking a CGI script (POST method)



7.4.4 Invoking a Servlet

You can invoke a servlet from a MIDlet the same way you would invoke a CGI script: by opening an HTTP connection and obtaining input/output streams on that connection. This section presents two examples.

- The first example invokes a servlet using the `GET` operation and collects and displays the results.
- In the second example, the servlet accepts input obtained from the user of the handset and invoked with the `POST` method.

7.4.4.1 FirstMidletServlet

In this example, `InvokeServletMidlet1` is invoked with the `GET` method and the response is received and displayed on the handset. No input is sent to the servlet. When invoked, the servlet sends the "Servlet Invoked!" string and the date back to the client. The source code for the `InvokeServletMidlet1` is in [Example 7-4](#).

Example 7-4. Invoking a servlet with no input values

```

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

```

```

import javax.microedition.midlet.*;

/**
 * An example MIDlet to invoke a servlet.
 */

public class InvokeServletMidlet1 extends MIDlet {

    private Display display;

    String url =
"http://127.0.0.1:8080/examples/servlet/HelloServlet";

    public InvokeServletMidlet1( ) {
        display = Display.getDisplay(this);
    }

    /**
     * Initialization. Invoked when we activate the MIDlet.
     */
    public void startApp( ) {
        try {
            invokeServlet(url);
        } catch (IOException e) {
            System.out.println("IOException " + e);
            e.printStackTrace( );
        }
    }

    /**
     * Pause, discontinue ....
     */
    public void pauseApp( ) {
    }

    /**
     * Destroy must cleanup everything.
     */
    public void destroyApp(boolean unconditional) {
    }

    /**
     * Retrieve a grade....
     */
    void invokeServlet(String url) throws IOException {
        HttpURLConnection c = null;
        InputStream is = null;
        StringBuffer b = new StringBuffer( );
        TextBox t = null;
        try {
            c = (HttpURLConnection)Connector.open(url);
            c.setRequestMethod(HttpURLConnection.GET);
            c.setRequestProperty("User-Agent", "Profile/MIDP-1.0
Configuration/CLDC-1.0");
            c.setRequestProperty("Content-Language", "en-CA");
            is = c.openDataInputStream( );
            int ch;
            while ((ch = is.read( )) != -1) {
                b.append((char) ch);
            }
        }
    }
}

```

```

        t = new TextBox("First Servlet", b.toString( ), 1024, 0);
    } finally {
        if(is!= null) {
            is.close( );
        }
        if(c != null) {
            c.close( );
        }
    }
    display.setCurrent(t);
}
}

```

The source code for the `HelloServlet`, which sends the message "Servlet Invoked!" and the date back to the client, is shown in [Example 7-5](#). You will need a web server that is capable of running servlets to make this work, such as the freely distributed Apache Tomcat.

Example 7-5. HelloServlet

```

import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

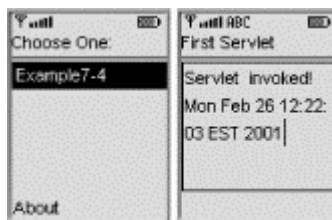
/**
 * The simplest possible servlet.
 */

public class HelloServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse
response)
        throws IOException, ServletException {
        response.setContentType("text/plain");
        PrintWriter out = response.getWriter( );
        out.println("Servlet Invoked!");
        out.println(new Date( ));
    }
}

```

When you run and activate the `InvokeServletMidlet1`, you should see something similar to [Figure 7-7](#).

Figure 7-7. FirstServletMidlet output



7.4.4.2 SecondMidletServlet

Now let's see how to invoke a servlet that expects input with the `POST` method. This is a more sophisticated example than the previous one. In this example, the

InvokeServletMidlet2 prompts the user to enter a value (the first name). When the user presses the key that corresponds to the `Submit` command, the `RequestServlet` is invoked. `RequestServlet` then retrieves the input values of the request from the buffer and returns them back to the client, showing that the servlet received the `POST` request. Note that the servlet and the MIDlet in this example run on the same machine. The source code for the `InvokeServletMidlet2` is shown in [Example 7-6](#).

Example 7-6. Invoking a servlet with an input value

```
import javax.microedition.rms.*;
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;
import javax.microedition.io.*;
import java.io.*;
import java.util.Vector;

public class InvokeServletMidlet2 extends MIDlet implements
CommandListener {
    Display display = null;
    List menu = null;
    TextBox input = null;
    String user = null;

    // note that the servlet and MIDlet run on the same machine
    String url =
        "http://127.0.0.1:8080/examples/servlet/RequestServlet2";
    static final Command backCommand = new Command("Back",
        Command.BACK, 0);
    static final Command submitCommand = new Command("Submit",
        Command.OK, 2);
    static final Command exitCommand = new Command("Exit",
        Command.STOP, 3);
    String currentMenu = null;

    public InvokeServletMidlet2( ) {
    }

    public void startApp( ) throws MIDletStateChangeException {
        display = Display.getDisplay(this);
        menu = new List("Invoke Servlet", Choice.IMPLICIT);
        menu.append("Add a user", null);
        menu.addCommand(exitCommand);
        menu.setCommandListener(this);
        mainMenu( );
    }

    public void pauseApp( ) {
    }

    public void destroyApp(boolean unconditional) {
        notifyDestroyed( );
    }

    void mainMenu( ) {
        display.setCurrent(menu);
    }

    public void addName( ) {
        input = new TextBox("Enter first name:", "", 5,
```

```

        TextField.ANY);
input.addCommand(submitCommand);
input.addCommand(backCommand);
input.setCommandListener(this);
input.setString("");
display.setCurrent(input);
}

void invokeServlet(String url) throws IOException {
    HttpURLConnection c = null;
    InputStream is = null;
    OutputStream os = null;
    StringBuffer b = new StringBuffer( );
    TextBox t = null;
    try {
        c = (HttpURLConnection)Connector.open(url);
        c.setRequestMethod(HttpURLConnection.POST);
        c.setRequestProperty("CONTENT-TYPE",
            "application/x-www-form-urlencoded");
        c.setRequestProperty("User-Agent",
            "Profile/MIDP-1.0 Configuration/CLDC-1.0");
        c.setRequestProperty("Content-Language", "en-CA");

        os = c.openOutputStream( );
        String str = "name="+user;
        byte postmsg[] = str.getBytes( );
        System.out.println("Length: "+str.getBytes( ));
        for(int i=0;i<postmsg.length;i++) {
            os.write(postmsg[i]);
        }
        // or you can easily do:
        //os.write(("name="+user).getBytes( ));
        os.flush( );

        is = c.openDataInputStream( );
        int ch;
        while ((ch = is.read( )) != -1) {
            b.append((char) ch);
            System.out.print((char)ch);
        }
        t = new TextBox("Second Servlet", b.toString( ), 1024, 0);
        t.addCommand(backCommand);
        t.setCommandListener(this);
    } finally {
        if(is != null) {
            is.close( );
        }
        if(os != null) {
            os.close( );
        }
        if(c != null) {
            c.close( );
        }
    }
    display.setCurrent(t);
}

public void commandAction(Command c, Displayable d) {
    String label = c.getLabel( );
    if (label.equals("Exit")) {
        destroyApp(true);
    }
}

```

```

    } else if (label.equals("Back")) {
        mainMenu( );
    } else if (label.equals("Submit")) {
        user = input.getString( );
        try {
            invokeServlet(url);
        } catch (IOException e) {}
    } else {
        addName( );
    }
}
}

```

The source code for the `RequestServlet`, which retrieves the `POST` request from the buffer and sends the input values back to the client, is shown in [Example 7-7](#).

Example 7-7. RequestServlet

```

import java.io.*;
import java.text.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
/**
 * Example servlet showing request headers
 */

public class RequestServlet extends HttpServlet {
    public void doPost(HttpServletRequest request, HttpServletResponse
response)
        throws IOException, ServletException {
        response.setContentType("text/plain");
        PrintWriter out = response.getWriter( );
        BufferedReader br = request.getReader( );
        String buf = br.readLine( );
        out.print("Rec: "+buf);
    }
}

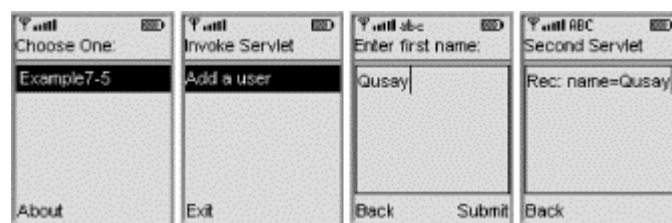
```



In order to use `ServletRequest.getParameter(String)` to retrieve the input values from the MIDlet in [Example 7-6](#), the `CONTENT_TYPE` header must be set to `application/x-www-form-urlencoded`, as discussed earlier in this chapter.

When you run `InvokeServletMidlet2` and invoke it by entering your name, you should see something similar to [Figure 7-8](#).

Figure 7-8. SecondMidletServlet output



7.5 Wireless Session Tracking

The term *session tracking* means maintaining state information about a series of requests from the same client. Maintaining such information for clients that use HTTP is a problem. Why? Because HTTP is a request-response protocol, which means the connection between the client and the server is not maintained for the duration of the conversation. In other words, HTTP is a *stateless* protocol. This means you cannot depend on the underlying connection protocol to maintain state information; you must find other ways to perform session tracking.

The two most widely used techniques for session tracking are *cookies* and *URL rewriting*. A cookie is a piece of data that a Web server sends to the client. This piece of data is stored by the client and used the next time the client makes a request from that server. However, if cookies are disabled by the browser or, more importantly, if the browser itself does not support them (as is the case with most current wireless devices), then cookies are not of much use. However, you can send and receive cookies through the use of the `HttpConnection.setRequestProperty()` and `HttpConnection.getHeaderField()` methods. To send a cookie to a server, simply set the value of the `cookie` request property before sending the message.

```
String myLocalCookieVariable;  
  
HttpConnection connection = (HttpConnection)Connection.open(someURL);  
Connection.setRequestProperty("cookie", myLocalCookieVariable);
```

When you receive a response back from the server, you can parse the resulting `Set-cookie` header field as follows:

```
is = c.openInputStream();  
String cookie = connection.getHeaderField("Set-cookie");  
If (cookie != null)  
    myLocalCookieVariable = cookie.substring(0, cookie.indexOf(";"));
```

The other technique for session tracking is URL rewriting. This technique is ideal for clients that do not support cookies or have cookies disabled. With this technique, the session information is encoded into URLs that the server generates. This means that instead of returning this URL (`http://www.somesite.com/servlet/shop/catalog.html`), the server generates the following URL, or something similar, instead (`http://www.somesite.com/servlet/shop/catalog.html;jsessionid=cl98373673At`).

The information that would otherwise be stored in a cookie is appended to the URL. You can parse this information out into a `String` after it arrives, using a technique similar the one we used with cookies in the previous section. The server will look for this information when a request is made from the client. The exact syntax of the encoded URL depends on the underlying server environment. However, the Java Servlet API provides facilities such as the `response.encodeURL(String)` and `response.encodeRedirectURL(String)` methods, which you can use for session tracking.

7.6 MIDlet Networking Security

Over the past few years, concerns about security on the Internet have heated up immensely. It's common in this day and age to hear of companies whose data has been hacked and from whom valuable credit card information has been stolen. It probably won't be long before people can "sniff" the data going through cell phones just as easily as data traveling across the Internet. What can you do? Fortunately, computers these days are capable of using *encryption* techniques to scramble data as it travels through the unsafe corridors of the Internet. In fact, one of the most popular forms of encryption uses a wide variety of cryptographic techniques to protect your data. It's called Secure Sockets Layer (SSL) and is built into practically every web browser.

Will you need encryption for your MIDlet programs? That's a hard question to answer. Many wireless protocols already use a sophisticated level of scrambling—far more than an average "sniffer" can decode. And while cryptographic software can be relatively small and easy to use on a desktop, using it on a cell phone can quickly expend both your processing power and program space. Understanding cryptography can take a bit of time as well.^[2] However, if you absolutely must have security on your cell phone to protect data traveling on the Internet, we recommend checking out the open source lightweight API software from "The Legion of the Bouncy Castle" (<http://www.bouncycastle.org>).

^[2] A great place to start, however, is Bruce Schneier's *Applied Cryptography*, Second Edition (Wiley).

Chapter 8. Database Programming

A database is a non-volatile place for storing the state of objects. For some applications, you might need objects to exist even after the application that created them closes. Without a storage mechanism, objects and their states are destroyed when an application closes. However, if you save objects to a *persistent storage facility*, such as a database, they can be read in later by the same application or even other applications.

The persistent storage facilities provided in the J2SE platform, such as the JDBC and Object Serialization APIs, are too large for handheld devices with a small memory footprint. Storage requirements vary significantly from one resource-constrained device to another. J2ME and the MIDP solve this problem by using the Record Management System (RMS).

This chapter introduces the details of the MIDP RMS, a persistent storage facility for MIDlets, and shows you how to develop MIDP database applications, using an example involving stocks. Throughout this chapter, the terms *record store* and *database* are used interchangeably.

8.1 The Record Management System

An RMS database (or record store) consists of a collection of records that remain persistent after the MIDlet closes. When you invoke the MIDlet again, it can retrieve data from the persistent record store. However, to use the RMS, we need to get familiar with some of the classes and concepts provided by the `javax.microedition.rms` package.