CHAPTER 5:

TESTING
Search Techniques (PhoneBook Codes Testing)

Searching with RecordFilter

To sorting the records (using RecordComparator), an enumerator can filter records. There is a subtle difference, however. When using RecordComparator all records in a record store are in the result set. With a RecordFilter, only those records that match the filter criteria will become part of the enumerator result set.

Following is a class that implements the RecordFilter interface.

```java
class SearchFilter implements RecordFilter {
}
```
private String searchText = null;

public SearchFilter(String searchText) {

    // This is the text to search for
    this.searchText = searchText.toLowerCase();
}

public boolean matches(byte[] candidate) {

    String str = new String(candidate).toLowerCase();
    // Look for a match
    if (searchText != null && str.indexOf(searchText) != -1) {
        return true;
    } else {
        return false;
    }
}

When an instance of this class is created we save the desired search text in a private variable. This text is then used when the enumerator calls the matches() method, passing in a record from the record store. Here is a small piece of code that shows how you create a reference between the filter and an enumerator.

// Create a new search filter
SearchFilter search = new SearchFilter("search text");

// Reference the filter when creating the result set
RecordEnumeration re = rs.enumerateRecords(search, null, false);

// If there is at least one record in result set, a match was found
if (re.numRecords() > 0) {
    // Do something
}
Example: Simple String Search

Two examples follow that use a RecordFilter to search the record store. Both MIDlets display a Form and a provide a TextField for entering the desired search string. The results of the first MIDlet are shown in Figure 5.1.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`boolean matches(byte[]</td>
<td>Search a record for a specific value</td>
</tr>
<tr>
<td>candidate)`</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1. RecordFilter Interface: javax.microedition.rms.RecordFilter

The reason for two examples is to keep a consistent pace with other examples we've created in this part. The first will write/search records that contain only pure text, which we
accomplish by writing a String object into each record. The second example will use streams to write/search the record store. In this case, each record will consist of a String, a boolean and an int. This example is probably a little more realistic in the sense that it will more closely mimic what types of data you may actually store. In addition, it proves to be a little more thought-provoking.

Example: SimpleSearch

```java
public void writeTestData()
{
    String[] golfClubs = {
        "Wedge...good from the sand trap",
        "One Wood...off the tee",
        "Putter...only on the green",
        "Five Iron...middle distance"};
    writeRecords(golfClubs);
}

public void writeRecords(String[] sData)
{
    byte[] record;
    try
    {
        // Only add the records once
        if (rs.getNumRecords() > 0)
            return;
        for (int i = 0; i < sData.length; i++)
        {
            record = sData[i].getBytes();
        }
    }
}
```
rs.addRecord(record, 0, record.length);
}
}
catch (Exception e)
{

dl(c.toString());
}

try
{
  // Record store is not empty
  if (rs.getNumRecords() > 0)
  {
    // Setup the search filter with the user requested text
    SearchFilter search =
    new SearchFilter(tfFind.getString());
    RecordEnumeration re =
    rs.enumerateRecords(search, null, false);
    // A match was found using the filter
    if (re.numRecords() > 0)
    // Show match in the stringItem on the form
    siMatch.setText(new String(re.nextRecord()));
    re.destroy(); // Free enumerator
  }
}
catch (Exception e)
```java
private void db(String str)
{
    System.err.println("Msg: \" + str + ");
}

/*
* Search for text within a record
* Each record passed in contains only text (String)
*---------------------------------------------------------------------*/
class SearchFilter implements RecordFilter
```
private String searchText = null;
public SearchFilter(String searchText)
{
    // This is the text to search for
    this.searchText = searchText.toLowerCase();
}

public boolean matches(byte[] candidate)
{
    String str = new String(candidate).toLowerCase();
    // Look for a match
    if (searchText != null && str.indexOf(searchText) != -1)
        return true;
    else
        return false;
}
At the top of the example you'll see the standard stuff—allocating the components and setting up the event listener. There are also a few lines for writing records to the record store. The text written to the store is simply an array of strings:

```java
String[] golfClubs = {
  "Wedge...good from the sand trap",
  "One Wood...off the tee",
  "Putter...only on the green",
  "Five Iron...middle distance"};
```

Before writing the searchFilter class that implements the RecordFilter interface, let's see how our enumerator in this example differs from the previous example.
// Setup the search filter with the user requested text

SearchFilter search = new SearchFilter(tfFind.getString());

RecordEnumeration re = re.enumerateRecords(search, null, false);

// A match was found using the filter
if (re.numRecords() > 0)
   _siMatch.setText(new String(re.nextRecord()));

First, we must allocate an instance of SearchFilter. Notice the constructor has a parameter that accepts the user-requested search text. Next, we create an enumerator with a reference to this class. If the result set of the enumeration has more than zero records, we know there was at least one match and we set the StringItem text.

At the bottom of the example is where you'll find the SearchFilter class. Notice the constructor saves the search text that was requested by the user. This text is then available to the method matches(), which is called by the enumerator during its initialization. This method will be called once for each record in the record store. Only those records that match the filter will be included in result set.

```java
public SearchStreams()
{
    display = Display.getDisplay(this);

    // Define textField, stringItem and commands
    tfFind = new TextField("Find", "", 10, TextField.ANY);
    siMatch = new StringItem(null, null);
    cmExit = new Command("Exit", Command.EXIT, 1);
```
cmFind = new Command("Find", Command.SCREEB, 2);

// Create the form, add commands
fmMain = new form("Record Search");
fmMain.addCommand(cmExit);
fmMain.addCommand(cmFind);

// Append textfield and stringItem
fmMain.append(tfFind);
fmMain.append(siMatch);

// Capture events
fmMain.setCommandListener(this);

//...-

// Open and write to record store

openRecStore(); // Create the record store
writeTestDaa(); // Write a series of records

public void destroyApp( boolean unconditional )
{

closeRecStore(); // Close record store
}
public void startApp()
{
displaysetCurrentFromMain();

public void pauseApp()
{
}

public void openRecStore()
{
    // See Example 11–1
}

public void closeRecStore()
{
    // See Example 11–1
}

/*..................................................................................
* Create three arrays to write into record store
*................................................................................../

public void writeTestData()
{
    String[] pets = {"duke - big, goofy golden retriever",
                     "tiger - we found in a field",
                     "spike - loves chasing car tires",
                     "beauregard - barks at everything");

    boolean[] dog = {true, false, true, true};

    int[] rank = {3, 0, 1, 2};

    writeStream(pets, dog, rank);
}

/*..................................................................................
* Write to record store using streams.
*................................................................................../

public void writeStream(String[] sData, boolean[] bData,
int[] iData)
try {
    // Only add the records once
    if (rs.getNumRecords() > fl)
        return;
    // Write data into an internal byte array
    ByteArrayOutputStream strmBytes =
        new ByteArrayOutputStream();
    // Write Java data types into the above byte array
    DataOutputStream strmDataType =
        new DataOutputStream(strmBytes);
    byte[] record;
    for (int i = fl; i < sDataLength; i++)
        { 
        // Write Java data types
        strmDataType.writeUTF(sData[i]);
        strmDataType.writeBoolean(bData[i]);
        strmDataType.writeInt(iData[i]);
        // Clear any buffered data
        strmDataType.flush();
        // Get stream data into byte array and write record
        record = strmBytes.toByteArray();
        rs.addRecord(record, 0, record.length);
        // Toss any data in the internal array so writes
        // starts at beginning (of the internal array)
        strmBytes.reset();
        }
    strmBytes.close();
    strmDataType.close();
}
try {
    if (rs.getNuniRecords() > 0) {
        SearchFiiter search = new SearchFiiter(tfFind.getString());
        RecordEnumeration re = rs.enumerateRecords(search, null, false);
        if (re.numRecords() > 0) {
            // Read from the specified byte array
            ByteArraylnputStream strmBytes = new ByteArraylnputStream(re.nextRecord());
            // Read Java data types from the above byte array
            DatalnputStream strmDataType = new DatalnputStream(strmBytes);
            // Show matching result in stringItem component on form
            siMatch.setText(strmDataType.readUTF());
        }
    }
}
} catch (Exception e) {
    db(e.toString());
}
search.searchFilterClose(); // Close record filter
stream.close(); // Close stream
streamDataType.close(); // Close stream
re.destroy(); // Free enumerator
}
}
}
}

catch (Exception e)
{

db(e.toString());
}

public void commandAction(Command c, Displayable s)
{
  if (c == cmdFind)
  {
    searchRecordStore();
  }
  else if (c == cmdExit)
  {
    destroyApp(false);
    notifyDestroyed();
  }
}

/*-----------------------------*/
/* Simple message to console for debug/errors */
/* When used with Exceptions we should handle the */
/* error in a more appropriate manner. */
/*-----------------------------*/

private void db(String str)
{

System.err.println("Myg: " + str);
}
}

/*---------------------------------------------
 * Search for text within a record
 * Each record passed in contains multiple Java data
 * types (String, boolean and integer)
 */
class SearchFilter implements RecordFilter {
{
    private String searchText = null;
    // Read from a specified byte array
    private ByteArrayInputStream strmBytes = null;
    // Read Java data types from the above byte array
    private DatalnputStream strmDataType = null;
    public SearchFilter(String searchText)
    {
        // This is the text to search for
        this.searchText = searchText.toLowerCase();
    }
    // Cleanup
    public void searchFilterClose()
    {
        try
        {
            if (strmBytes != null)
                strmBytes.close();
            if (strmDataType != null)
                strmDataType.close();
        }
        catch (Exception e)
public boolean matches(byte[] candidate) {
    String str = null;
    try {
        strmBytes = new ByteArrayInputStream(candidate);
        strmDataType = new DataInputStream(strmBytes);
        // Although 3 pieces of data were written to
        // the record (String, boolean and integer)
        // we only need one read because the string to
        // search is the first "field" in the record
        str = strmDataType.readUTF().toLowerCase();
    } catch (Exception e) {
        return false;
    }
    // Look for a match
    if (str != null && str.indexOf(searchText) != -1) {
        return true;
    } else {
        return false;
    }
}
Figure 5.3. First Menu of PhoneBook Screen shot

Once again, the basics are the same: set up the Form, TextField and Command objects. When writing to the record store, we use streams to write each array element as a primitive Java data type, for example:

```java
strmDataType.writeUTF(sData[i]); // Write Strings
strmDataType.writeBoolean(bData[i]); // Write booleans
strmDataType.writeInt(iData[i]); // Write integers
```

Inside searchRecordStore() is where we create the search filter and enumerator. Before going into the specifics of this method, let's review the code inside SearchFilter, the class that implements the RecordFilter interface.

As with the previous example, we save the search text into a local variable:

```java
this.searchText = searchText.toLowerCase();
```
The method matches() is called by the enumerator for each record in the record store. To get the data in the same format as it was written to the record, we need to create two streams—one that references the byte array with the incoming data and another to read Java data types from the previous stream:

```java
strmBytes = new ByteArrayInputStream(candidate);
strmDataType = new DataInputStream(strmBytes);
```

Although we've written three pieces of data into each record (String, boolean, int) our search only applies to the first "field," the String. So we extract the String and store it in a local variable:

```
str = strmDataType.readUTF().toLowerCase();
```

![Figure 5.4. Adding, Sorting and Help by using TextField Screen shot](image)

The remainder of the code is straightforward. Compare the text from the record with the text in the variable searchText, which we saved when we created the filter:
if (str != null && str.indexOf(searchText) != -1)
    return true;
else
    return false;

We can finish this discussion by looking back at searchRecordStore(). If the enumerator has more than 0 records as part of the result set, we know there was a successful match between the text the user requested and one or more records in the record store:

if (re.numRecords() > 0)
{
    // Read from the specified byte array
    ByteArrayInputStream strmBytes =
        new ByteArrayInputStream(re.nextRecord());
    // Read Java data types from the above byte array
    DataInputStream strmDataType =
        new DataInputStream(strmBytes);
    // Show matching result in stringItem component on form
    siMatch.setText(strmDataType.readUTF());
    ...
}

Before we can display the text from the record (on the form), once again we need to extract the data in the same format in which it was written. We do this by setting up the necessary
streams, reading the String from the record and adding the resulting text into the StringItem on the form.

### Exception Handling

There are five exceptions that are specific to the RMS. As expected, each is inherited from the class `java.lang.Throwable`. All five exceptions are listed next, along with a brief description and the constructor methods in the event your application code needs to create/throw an RMS-related exception.

- **InvalidRecordIDException**: Used to indicate an invalid record number. For example, if `RecordStore.getRecord(int)` is called with a record that does not exist in the record store, this exception will be thrown.

  ```java
  public InvalidRecordIDException(String message)
  public InvalidRecordIDException()
  ```

- **RecordStoreException**: A general exception indicating an error with the record store.

  ```java
  public RecordStoreException()
  public RecordStoreException(String message)
  ```
Signals the record store is full. For example, if `RecordStore.add Record(byte[], int, int)` is called to add a new record, this exception will be thrown if the record store has reached its maximum capacity.

```java
public RecordStoreFullException()
public RecordStoreFullException(String message)
```

Indicates the record store name does not exist. For example, calling `RecordStore.deleteRecordStore(String)` with an invalid record store name will result in this exception being thrown.

```java
public RecordStoreNotFoundException()
public RecordStoreNotFoundException(String message)
```

Used to indicate that the requested record store is not open. For example, if `RecordStore.getNumRecords()` is called prior to opening the record store, this exception will be thrown.

```java
public RecordStoreNotOpenException()
public RecordStoreNotOpenException(String message)
```
To keep the examples in this part focused on the concepts presented, I've done the absolute minimum to appease the Java compiler when dealing with exceptions. Typical code has simply caught every type of exception and written a message to the console:

```java
RecordStore rs;
try {
   rs.closeRecordStore();
} catch (Exception e) {
   db(e.toString());
}
...
private void db(String str)
{
   System.err.println("Msg: " + str);
}
```

Many record store methods throw exceptions. Any production application will need to implement a more robust means of dealing with problems as they arise. All exceptions should be caught and handled in an appropriate manner, depending on the type of error. Ideally, the aforementioned code should look a little more like the following:
try {

    rs.closeRecordStore();

} catch (RecordStoreNotOpenException e) {
    do something here...

} catch (RecordStoreException e2) {
    do something else here...

} catch (Exception e3) {
    other error condition...
}

The missing code ("do something here") will be dependent on the error that occurred. For example, if the record store was not open, it may be as simple as calling a method to open the record store and retrying the operation. If the error is more on the esoteric side (as with the RecordStoreException), you may need to notify the user and ask how they would like to proceed.
Weighing in at nearly 200 kilobytes, the 100+ classes and interfaces in J2SE java.io and java.net will exceed the resources available on many mobile devices. Beyond memory considerations, mobile devices will span the gamut in support for network protocols and file systems. To provide an extensible framework for i/o and networking, the GCF was developed.

The goal was not to devise a completely new set of classes, but rather to provide a subset of J2SE that is more in tune with the limitations and variations likely to be found on devices implementing MIDP.

The basic idea is to have one class, the Connector, that can create any type of connection, file, http, datagram, and so forth. The open method has the following form:

```java
Connector.open("protocol:address:parameters");
```

For example,

```java
Connector.open("http://www.some_web_address.com");
Connector.open("socket://someaddress:1234");
Connector.open("file://testdata.txt");
```

How the protocols are resolved is where the flexibility of the GCF comes into play. At runtime, Connector looks for the appropriate class that implements the requested protocol.
This is done using Class.forName(). A request to open a HTTP connection in J2ME (using Sun's implementation) may look as follows:

```java
Class.forName("com.sun.midp.io.J2me.HttpProtocol");
```

If a class is found, an object is returned that implements a Connection interface. The Connector class and Connection interfaces are defined in CLDC. Figure 5.5 shows the classes that make up the Connection hierarchy, where each class is defined as an interface.

The actual implementation of the protocol(s) is at the Profile level. For example, in MIDP 1.0 the class HttpConnection supports a subset of HTTP version 1.1. In Figure 5.5, notice
HttpConnection extends ContentConnection and in turn provides over 20 methods for working specifically with HTTP.

Although DatagramConnection is included in the hierarchy, an implementation of MIDP is not required to support this protocol. We'll focus on HTTP, as it is the only requirement of MIDP.

Before moving on, let's have a closer look at the definitions of the classes and methods in the GCF.

 Hopefully this will provide some insight as you work through the examples that follow.

```java
Connection (public abstract interface Connection)
    public void close()

InputConnection (public abstract interface InputConnection extends Connection)
    public InputStream openInputStream()
    public DataInputStream openDataInputStream()

OutputConnection (public abstract interface OutputConnection extends Connection)
    public OutputStream openOutputStream()
    public DataOutputStream openDataOutputStream()

StreamConnection (public abstract interface StreamConnection extends InputConnection, OutputConnection)

ContentConnection (public abstract interface
```
ContentConnection extends Stream

Connection)

public long getLength()

public String getEncoding()

public String getType()

HttpConnection (public interface HttpConnection

extends ContentConnection)

20+ methods for communicating over HTTP

Connector (public class Connector)

public static Connection open(String name)

public static Connection open(String name, int mode)

public static Connection open(String name, int mode, boolean
timeouts)

public static DataInputStream openDataInputStream(String name)

public static DataOutputStream openDataOutputStream(String name)

public static InputStream openInputStream(String name)

public static OutputStream openOutputStream(String name)
In MIDP 1.0 the only protocol that is guaranteed to be implemented is http. Through the class HttpConnection you can communicate with a web server or any remote device that supports HTTP.

HTTP is known as a request/response protocol. A client initiates a request, sends it to a server with an address specified as a Uniform Resource Locator (URL), and a response is returned from the server.

This is how a web browser and web server work together. You type in the URL (e.g., www.sun.com) and then the web browser creates a package of information (more on this in a moment), sends the request out over the network, receives information from a web server and displays the result in the browser.

The Connector class has seven methods to create a connection with a server (see Table 5.2). There are three variations of the open() method. The first requires only the address of the server. The second method accepts a mode for reading/writing (see Table 5.3). The third option includes a boolean flag that indicates if the caller of the method can manage timeout exceptions. The remaining methods open various input and output streams.

Here is a call to create a ContentConnection. Refer back to Figure 5.5 to see the Connection hierarchy.

```java
// Create a ContentConnection
String url = "http://www.sun.com"
ContentConnection connection =
    (ContentConnection) Connector.open(url);
```
One option to read data over this connection is through an InputStream:

```java
InputStream isStream = connection.openInputStream();
// ContentConnection includes a length method
int length = (int) connection.getLength();
if (length > 0)
{
    byte imageData[] = new byte[length];
    // Read the data into an array
    isStream.read(imageData);
}
```

<table>
<thead>
<tr>
<th>Method (each declared static)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static Connection open(String name)</td>
<td>Create connection in READ_WRITE mode</td>
</tr>
<tr>
<td>static Connection open(String name, int mode)</td>
<td>Create connection with specified mode</td>
</tr>
<tr>
<td>static Connection open(String name, int mode, boolean handlesIOExcs)</td>
<td>Create connection with specified mode, handling IO exceptions</td>
</tr>
<tr>
<td>static InputStream openInputStream(String name)</td>
<td>Create connection input stream</td>
</tr>
<tr>
<td>static OutputStream openOutputStream(String name)</td>
<td>Create connection output stream</td>
</tr>
<tr>
<td>static DataInputStream openDataInputStream(String name)</td>
<td>Create connection data input stream</td>
</tr>
<tr>
<td>static DataOutputStream openDataOutputStream(String name)</td>
<td>Create connection data output stream</td>
</tr>
</tbody>
</table>

Table 5.2. CLDC Connector Methods: javax.microedition.io.Connector

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ</td>
<td>Open connection for read only</td>
</tr>
<tr>
<td>WRITE</td>
<td>Open connection for write only</td>
</tr>
<tr>
<td>READ_WRITE</td>
<td>Open connection for read and write</td>
</tr>
</tbody>
</table>

Table 5.3. CLDC Connector Modes: javax.microedition.io.Connector
The code creates a connection with a server by specifying a URL, opens an InputStream and reads data from the remote server into an array. For this example, there is no need to send or receive HTTP commands.

For the sake of argument, let's create an InputStream connection and bypass ContentConnection altogether. As you'll see, the obvious loss with this approach is no method to determine the length of the data:

```java
// Create an InputStream connection

String url = "http://www.sun.com/sun1.png"

InputStream iStrm = (InputStream) Connector.openInput(url);

try
{
  byte imageData[] = new byte[2500];

  // Read the data into an array

  int length = iStrm.read(imageData);
}
```

When we get to the first example, we'll see how to use a ByteArrayOutputStream to take the guesswork out of determining how large an array to allocate if the length is not known.

Although you may find yourself opening many types of connections (the aforementioned are just a sample), the heart and soul of this chapter is devoted to HttpConnection. Here's how we may open such a connection:

```java
String url = "https://www.sun.com/sun1.png"

HttpConnection http = (HttpConnection) Connector.open(url);
```
Once open, this connection offers access to various streams (InputStream, DataInputStream, to name a few). However, as we'll see, the advantage to using this connection lies with the HTTP commands at our disposal.

The key point to hold onto is this: Regardless of the type of connection, whether it is ContentConnection, HttpConnection or otherwise, communication with a server always begins with a call to one of the seven methods in the Connector class, Table 5.2.

**Example: Download and View with InputStream**

To show the flexibility of the Connection interface, let's change the previous example to download the same data with an InputStream. We need to change only a few lines of code in the method getImage(). Here is the updated version:

```java
private Image getImage(String url) throws IOException
{
```
InputStream iStrm = (InputStream) Connector.openInputStream(url);

Image im = null;

try {

ByteArrayOutputStream bStrm = new ByteArrayOutputStream();

int ch;

while ((ch = iStrm.read()) != -1)
    bStrm.write(ch); // Place into image array

byte imageData[] = bStrm.toByteArray();

// Create the image from the byte array

im = Image.createImage(imageData, 0, imageData.length);

} finally {

// Clean up

if (bStrm != null)
    bStrm.close();

return (im == null ? null : im);
One drawback is the absence of a method to determine the length of the incoming data. However, this is not much of a concern, simply use a ByteArrayInputStream to read the data and move the results into the destination array.

Up to this point, we have not had a need to send or receive HTTP commands. The objective was simply to create a connection and download a stream of data. It's time to shift gears and learn how to create and manage communication through the HttpConnection class.

HTTP is referred to as a request/response protocol: client requests information, a server sends a response. The most common example is the interaction between a web browser (the client) and a web server. This section and the next will look at the details of the client request and the server response.

Request Method
A client request, known as the request entity, consists of three sections: request method, header and body. There are three request methods available using HttpConnection (see Table 5.4).

<table>
<thead>
<tr>
<th>Request</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Request information — data sent as part of URL.</td>
</tr>
<tr>
<td>POST</td>
<td>Request information — data sent in separate stream.</td>
</tr>
<tr>
<td>HEAD</td>
<td>Request only &quot;metadata&quot; about a resource.</td>
</tr>
</tbody>
</table>

Table 5.4. Request Method: javax.microedition.io.HttpConnection

All three request methods inform a server that a client is requesting some type of information. For GET and POST, what differs is how data from the client is transferred to the server. Both methods, along with HEAD, will be explained in a moment.

You specify the request method of an HttpConnection through setRequestMethod()
HttpConnection http = null;
http = (HttpConnection) Connector.open(url);
http.setRequestMethod(HttpConnection.GET);

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void setRequestMethod(String method)</td>
<td>Set the request method (GET, POST or HEAD)</td>
</tr>
<tr>
<td>void setRequestMethod(String key, String value)</td>
<td>Set a request property (header information)</td>
</tr>
<tr>
<td>String getRequestMethod()</td>
<td>Get the current setting of the request method (GET, POST, HEAD)</td>
</tr>
<tr>
<td>String getRequestProperty(String key)</td>
<td>Get the current setting of a request property (header value)</td>
</tr>
</tbody>
</table>

Table 5.5. Client Request Methods: java.awt.microedition.io.HttpConnection

Using GET, the body (data) of the request becomes part of the URL. This is known as URL encoding.

For example, using the analogy of a web browser, assume we have a form with two fields—color and font. The names of the fields on the form, as defined in the HTML code, are userColor and userFont. When the form has been completed and is ready to be sent, the values entered onto the form will be appended onto the URL.

Data submitted using POST is sent separately from the call to the URL. Put another way, the request to open a connection to a URL is sent as one stream, any data is sent as a separate stream. There are two major benefits of POST over GET:

1. POST has no limit to the amount of data that can be sent. This is not the case for the request method GET. When a server processes a GET request, the data from the end of the URL is stored in an environment variable (known as a query string). When sending a large amount of data, you run the risk of overrunning the environment variable.
2. POST sends data as a separate stream; therefore, the contents are not visible as part of the URL.

HEAD works in the same manner as GET, with the client sending the body as part of the URL. However, the server will not return a body in its response (more on the server response in a moment).

HEAD is generally used for retrieving information about a resource on a server. For example, you may want information about the last modified date of a file; however, it may not be necessary to retrieve the file contents.

Header Information
The second part of a client request is header information. The HTTP protocol defines over 40 header fields. Some of the more common are Accept, Cache-Control, Content-Type, Expires, If-Modified-Since and User-Agent. Headers are set by calling setRequestProperty().

```java
HttpConnection http = null;

http = (HttpConnection) Connector.open(url);

http.setRequestMethod(HttpConnection.GET);

http.setRequestHeader("If-Modified-Since",
"Mon, 16 Jul 2001 22:54:26 GMT");
```
Body

Data to be transferred from the client to the server is referred to as the body of the request. As mentioned, GET sends the body as part of the URL. POST sends the body in a separate stream.

Once a client has packaged together the request method, header and body, and sent it over the network, it is now up to the server to interpret the request and generate a response known as the response entity. As with the client request, a server response consists of three sections: status line, header and body.

Creating the response on the server side is beyond the scope of this discussion. Instead, we will concern ourselves with how to pull information from the server response.

Status Line

The status line indicates the outcome of the client request. For HttpConnection, there are over 35 status codes reported. HTTP divides the codes into three broad categories based on the numeric value mapped to the code.

1xx—information
2xx—success
3xx—redirection
4xx—client errors
5xx—server errors

For example, HTTP_OK (status code 200) indicates the client request was successfully processed.

When sending a response, a server includes the protocol version number along with the status code.

Here are a few sample status lines:
When interpreting status codes, you have two options: retrieve the message or the code.

```java
http.getResponseMessage();

http.getResponseCode();
```

The first returns only the text portion, such as OK or Bad Request. The later returns the code, such as 200 or 400.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int getResponseCode()</td>
<td>Get the response code (numeric)</td>
</tr>
<tr>
<td>String getResponseMessage()</td>
<td>Get the response message (text)</td>
</tr>
</tbody>
</table>

Table 5.6. Response Methods: javax.microedition.io.HttpConnection

**Header**

Like the client, the server can send information through a header. These key-value pairs can be retrieved in various shapes and forms through the methods in Table 5.7.
As an example, let's assume the server sent a header key-value pair content-type=text/plain. As a further assumption, the key-value pair will be the first entry in the header, and with this assumption, it will have an index value of 0. This header informs the client that the body of the response will be returned as plain text (as compared to HTML text, etc.). The following calls all reference the same key-value pair:

```java
// Header field at index 0: "content-type=text/plain"
http.getHeaderField(0); // "text-plain"
http.getHeaderField("content-type"); // "text-plain"
http.getHeaderFieldKey(0); // "content-type"
```

**Table 5.7. Server Response Methods: java.microedition.io.HttpConnection**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String getHeaderField(int n)</td>
<td>Get header field value looking up by index</td>
</tr>
<tr>
<td>String getHeaderField(String name)</td>
<td>Get header field value looking up by name</td>
</tr>
<tr>
<td>long getHeaderFieldDate(String name, long def)</td>
<td>Get named field as a long (representing the date)</td>
</tr>
<tr>
<td>int getHeaderFieldInt(String name, int def)</td>
<td>Get named field as an integer</td>
</tr>
<tr>
<td>String getHeaderFieldKey(int n)</td>
<td>Get header field key using index</td>
</tr>
<tr>
<td>long getDate()</td>
<td>Get header field &quot;date&quot;</td>
</tr>
<tr>
<td>long getExpiration()</td>
<td>Get header field &quot;expires&quot;</td>
</tr>
<tr>
<td>long getLastModified()</td>
<td>Get header field &quot;last-modified&quot;</td>
</tr>
</tbody>
</table>

**Body**

The body is the data sent from the server to the client. There are no methods defined in HttpConnection for reading the body.

**Example: ViewFile On HTTP Connection**

```java
private void processRequest() throws IOException
{
    HttpConnection http = null;
```
InputStream iStriim = null;
try {
    // Create the connection
    http = (HttpConnection) Connector.open(url);
    //----------------
    // Client Request
    //----------------
    // 1) Send request method
    http.setRequestMethod(HttpConnection.GET);
    // 2) Send header information (this header is optional)
    http.setRequestHeader("User-Agent", "Profile/MIDP-1.0 Configuration/CLDC-1.0");
    // http.setRequestHeader("If-Modified-Since", "Mon, 16 Mar 2008 22:54:26 GMT");
    // If you experience IO problems, try
    // removing the comment from the following line
    // http.setRequestHeader("Connection", "close");
    // 3) Send body/data - No data for this request
    //----------------
    // Server Response
    //----------------
    System.out.println("url: " + url);
    System.out.println("-----------");
    // 1) Get status Line
    System.out.println("Msg: " + http.getMessage());
    System.out.println("Code: " + http.getResponseCode());
    // 2) Get header information
    if (http.getResponseCode() == HttpConnection.HTTP_OK) {
        System.out.println("Field 0: " + http.getHeaderField(0));
        System.out.println("Field 1: " + http.getHeaderField(1));
System.out.println("field 2: " + http.getHeaderField(2));
System.out.println("------------------");
System.out.println("key 0: " + http.getHeaderField(0));
System.out.println("key 1: " + http.getHeaderField(1));
System.out.println("key 2: " + http.getHeaderField(2));
System.out.println("content: " + http.getHeaderField("content-type"));
System.out.println("date: " + http.getHeaderField("date"));
System.out.println("last-modified: " + http.getHeaderField("last-modified"));
System.out.println("------------------");

// 3) Get data (show the file contents)

String str;
InputStream inStrm = http.openInputStream();
int length = (int) http.getLength();
if (length != -1)
{
    // Read data in one chunk
    byte serverData[] = new byte[length];
    inStrm.read(serverData);
    str = new String(serverData);
}
else // Length not available...
{
    ByteArrayOutputStream bStrm =
        new ByteArrayOutputStream();
    // Read data one character at a time
    int ch;
    while ((ch = inStrm.read()) != -1)
        bStrm.write(ch);
str = new String(bsrm.toByteArray());
"Strm.close();
}
System.out.println("File Contents: " + str);

// Show connection information

System.out.println("Host: " + http.getHost());
System.out.println("Port: " + http.getPort());
System.out.println("Type: " + http.getType());
}
}
finally
{
// Clean up
if (bsrm != null)
bsrm.close();
if (http != null)
http.close();
}

cubic void pausoApp()
{
}
public void destroyApp(boolean unconditional)
{
}
Once a connection has been established, there are several methods available to obtain information about the connection. Each is listed in Table 5.8.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String getString()</td>
<td>Get filename from the URL</td>
</tr>
<tr>
<td>String getHost()</td>
<td>Get host from the URL</td>
</tr>
<tr>
<td>int getPort()</td>
<td>Get port from the URL</td>
</tr>
<tr>
<td>String getProtocol()</td>
<td>Get protocol from the URL</td>
</tr>
<tr>
<td>String getQuery()</td>
<td>Get the query string (only valid with a GET request)</td>
</tr>
<tr>
<td>String getRef()</td>
<td>Get the reference portion of URL</td>
</tr>
<tr>
<td>String getURL()</td>
<td>Get the entire URL</td>
</tr>
</tbody>
</table>

Table 5.8. Connection Methods: javax.microedition.io.HttpConnection

Server Response
Although this section won't give a lengthy tutorial on writing servlets, there will be enough information to clearly understand how a servlet interprets a client request and generates a response.

A servlet will run one of two methods, depending on whether the client request method is GET or POST.

```java
doGet(HttpServletRequest req, HttpServletResponse res)
doPost(HttpServletRequest req, HttpServletResponse res)
```

The parameter HttpServletRequest provides access to the client data that initiated the request to the servlet. HttpServletResponse is used to package a response for the client.
protected void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
    String acct = req.getParameter("account");
    String pwd = req.getParameter("password");

    String balance = accountLookup(acct, pwd);
    if (balance == null) {
        res.sendError(res.SC_BAD_REQUEST, "Unable to locate account");
        return;
    }
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();
    out.print(balance);
}

The first course of action is to look for parameters that may have been sent as part of the client header. These values will be stored in the variables acct and pwd. These values correspond to the textfields on the client form. With this information, the servlet can then search the database in an attempt to look up the account balance. If the search fails an error is returned to the client; otherwise, a response containing the balance is created.
The first line above creates a header field specifying the "content-type." This informs the client that the data from the server will be sent as plain text. Next, the servlet obtains a resource to write text to the client. The last steps are to output the balance and close the writer.

The code inside doPost() is identical. The reason for not consolidating the code is to illustrate the point that data can be sent using either GET or POST and processed in the same fashion on the server.

There is one last detail, and that is how the servlet looks up information in the account database.

```java
private String accountLookup(String acct, String pwd)
{

// These will vary depending on your server/database

Class.forName("sun.jdbc.odbc.Jdbc3ODBCDriver");

con = DriverManager.getConnection("jdbc:odbc:acctinfo");

Statement stmt = con.createStatement();

ResultSet rs = stmt.executeQuery("Select balance from acctInfo

where account = " + acct + " and password = " + pwd + "")

if (rs.next())

  return rs.getString(1);

else

  return null;

```
Through JDBC, a connection is established with the database. A simple Structured Query Language (SQL) statement will search the database. If the result set is not empty, return the account balance as a String.

Updating the Client
Now that the client has sent a request and the server has responded, the MIDlet needs to interpret the server reply and update the display with the account balance.

```java
processServerResponse(HttpConnection http, InputStream iStrm)
{
    // Reset error message
    errorMsg = null;
    // 1) Get status Line
    if (http.getResponseCode() == HttpConnection.HTTP_OK)
    {
        // 2) Get header information - none

        // 3) Get body (data)
        int length = (int) http.getLength();
        if (length > 0)
        {
            byte servletData[] = new byte[length];
            iStrm.read(servletData);
            // Update the string item on the display
        }
    }
    return null;
}
```
siBalance.setText(new String(servletData));
return true;
}
else
errorMsg = new String("Unable to read data");
}
else
// Use message from the servlet
errorMsg = new String(http.getResponseMessage());
return false;
}

We pull apart the server response in three chunks. First, we look at the status line to verify the server was able to fulfill the request. The next step is to read any header information, but there is none for this MIDlet. The body of the server response contains what we are after—the account balance. With this in hand we can update siBalance, the StringItem that shows the account balance on the form.
To have a single player game experience is the time to connect to rest of the world. After all better than show off our high-scores to our friends is to show them to everybody in the Internet!

One of the main features of mobile devices is the ability to be connected almost everywhere anytime. Java ME allows easy access to these communications features of your device:

- HTTP
- Socket
- SMS
- Bluetooth

During this lesson we are going to learn to use a HTTP connection to send our high scores data to a central server and to use the SMS functionality to invite other friends to play our game.

The main entry point for network functionality is the factory class Connector. This class allows the creation of Connection objects through the following static methods:

- open(String name)
- open(String name, int mode)

The name parameter is a URL string that identifies the connection you want to make. Depending of the protocol in the URL, a different kind of Connection object is returned. Here is a list of URL and the correspondent Connection objects:

- "http://www.server.com", returns a http connection, using the class HttpConnection
- "socket://server.com:8080", returns a TCP/IP connection
Of those protocols only HTTP is mandatory on all Java ME phones, all the others are device dependent, however the majority of new phones support all these protocols.

One thing you must be aware is that when you try to open any kind of connection, the user of your application is notified by the Application Management System. He then has the option to accept or deny your connection, if he denies your application will receive a SecurityException and no communication will happen.

Let's start with HTTP connections, one of the most common type of connections used. To create an HTTP connection you just need to do the following:

```java
public class Network {

    public byte[] httpConnection(String url, byte[] dataToSend)
    throws IOException {
        HttpConnection hc = null;
        // Prepare Connection
        hc = (HttpConnection) Connector.open(url, Connector.READ_WRITE);
        [...]
    }
}
```

The HTTP connections are represented by the HttpConnection class, this class has three states:
* Setup, in which you can setup the request parameters
* Connected, where you can send and receive data. You can only send data first and then receive the answer.
* Closed, after you read the data and the connection is ended.

When you receive the HttpConnection object from the Connector it's in the Setup state. You can then configure the request parameters. One of the main options is the HTTP request method (POST, GET or HEAD).

```java
public byte[] httpConnection(String url, byte[] dataToSend) throws IOException {
    ...
    if (dataToSend == null){
        hc.setRequestMethod(HttpConnection.GET);
    } else {
        hc.setRequestMethod(HttpConnection.POST);
    }
    ...
}
```

You can also configure the request properties like the content type of what you are sending in case of a POST.

```java
hc.setRequestProperty("Content-type", "application/octet-stream");
```

After you have configured the connection you can start send data, when you start this process you cannot change the request parameters. If the communication goes through you start receiving data.
Figure 5.7. Send the score to the Net(Internet) in Hit Brick Game, Screen shot

```java
public byte[] httpConnection(String url, byte[] dataToSend) throws IOException {
    // Write Data
    OutputStream os = hc.openOutputStream();
    os.write(dataToSend);
    os.close();
}

// gets answer from server
int rc = hc.getResponseCode();
// check http response
if (rc == HttpConnection.HTTP_OK){
```
We now have our `httpConnection()` method complete let's use it in our game to send the high scores to a server. Start by adding a new option to our highscores screen:

```java
public Displayable initHighscoreForm() {
    Command sendCommand = new Command("Send to Server", Command.ITEM, 1);
    highscoreForm.addCommand(sendCommand);
    // rest of method...
}
```

And then create a method to use it when this Command is selected:

```java
public String sendScore(String user, int score) {
    String result = "No Answer";
    // server to send data
    String url = "http://www.server/midp/post.php";
    // prepare http request
    String urlTotal = url + "?user=" + user + ";score=" + score;
    byte[] data = null;
    // rest of method...
}
```
```java
try {
    data = network.httpConnection(urlTotal, null);
} catch (IOException e) {
    result = "Communication Problems";
    e.printStackTrace();
} catch (SecurityException s) {
    // user denied access to communication
    result = "You need to allow communications in order to send the highscore to server."
    s.printStackTrace();
}
// check data return.
if (data != null) {
    result = new String(data);
}
return result;
```

And call it when the Command is used

```java
if (cmd == comInviteSend) {
    result = sendScore(scores[0].name, scores[0].value);
    display(new Alert("Result", result, null, AlertType.INFO));
}
```

If you try this code as it is you will receive the following warning:

Warning: To avoid potential deadlock, operations that may block, such as networking, should be performed in a different thread than the commandAction() handler.

Remember, you are executing this network code as an answer to an user event, that is being executed on the UI thread. If this code blocks or takes a long time to execute all the application will be stuck.
To avoid this issue you must use a thread to do you network related tasks. We start by declaring a thread and some state variables to control the actions of the thread.

```java
private static final int ACTION_SEND_HIGHSORE = 0;
public Thread workThread;
public boolean active = false;
```

Then every time we need to do a time consuming event, we call a method that prepares the thread and starts the action.

```java
public void doAction(int action) {
    // stores action to do
    this.action = action;
    // check if thread is already created
    if (workThread == null) {
        workThread = new Thread(this);
        workThread.start();
        active = true;
    }
    // wakes up thread to work
    synchronized (this) {
        notify();
    }
}
```

The new thread will execute our run method and do the actions we need.

```java
public void run() {
```
while (active) {
  
  // check what action to do

  switch (action) {

    case ACTION_SEND_HIGHSCORE:
      
      // send the first score to the server
      result = sendScore(scores[0].name, scores[0].value);

      commandAction(cmdReceiveHighScore, highScoreForm);

      break;

    }

  // waits for action to do.

  synchronized (this) {

    try {
      
      [-]

    } catch (InterruptedException e) {

      e.printStackTrace();

    }

  }

}

Don't forget to declare the Runnable interface in your midlet

public class MyMidlet extends MIDlet implements CommandListener, Runnable {

  Now we just need to call our doAction() method when the user chooses the send high score
  option. You may have noticed that our run() method invokes a command called
  cmdReceiveHighScore. This is used to show results of the communication to the user.

  if (cmd == cmdSendHighScore) {

Our second use to communications in JavaME is to send invitations to play the game to our friends. And you can have some competition in the high-scores tables.

First we are going to implement the method that sends the sms on our network class.

```java
public boolean sendSms(String number, String message) {
    boolean result = true;
    try {
        //set address to send message
        String addr = "smsc://" + number;
        // opens connection
        [...]
        msg.setPayloadText(message);
        // send message
        conn.send(msg);
    } catch (Exception e) {
        result = false;
    }
    return result;
}
```
As you can see we just need to use the open() method of the Connector class with an url of the format "sms://number". Then you create a TextMessage and send it trough the Connection. The size of the message is limited to around 160 characters.

Now we need to create a form to receive the name and phone information from the user.

```java
public Displayable inviteForm() {
    if (inviteForm == null) {
        inviteForm = new Form("Invite");
        inviteForm.setCommandListener(this);
        inviteForm.addCommand(initBackCommand());

        inviteName = new TextField("Name:", "", 20, TextField.ANY);
        inviteNumber = new TextField("Number:", "", 20, TextField.PHONE_NUMBER);
        inviteForm.append(inviteName);
    }

    return inviteForm;
}

And implement the commands action to show our form and send the message using our worker thread.

```
Finally we implement the new action in our run method

```java
public void run() {
    ...
    case (ACTION_SEND_INVITE):
        // invite another player to play
        String inviteMessage = " invites you to play Omid - Hit Brick!";
        ...
}
```

**Figure 5.8.** Invite and inform your friend to play the Hit Brick Game, Screen shot