J2ME crash course

Harald Holone

2006-01-24

Abstract

This article gives a short, hands-on introduction to programming J2ME applications on the MIDP 2.0 platform. Basic concepts, such as configurations, profiles and optional packages are covered. A few sample applications demonstrating the Midlet life cycle and user interface are provided. A demonstration of how to connect to a web server is given, and finally, we look at how to connect to other devices using Bluetooth [2] technology.

For a more in-depth introduction to MIDP programming, see one of the many books available [1].

1 Introduction

J2ME is a Java platform for devices with limited resources. There's a wast number of different devices with different capabilities out there, so J2ME is a platform described in the form of *configurations* and *profiles*. In addition there are more APIs that extend the basic capabilities of theses configurations and profiles.

1.1 CLDC

A configuration defines a Java API and minimum processing and memory requirements for a device implementing it. CLDC (Connected Limited Device Configuration) is one such profile, targeting devices such as mobile phones or PDAs. With a few exceptions, the API defined by CLDC is a subset of the familiar J2SE packages `java.lang`, `java.util` and `java.io`. All classes in the CLDC API that corresponds to classes in J2SE. In addition, CLDC defines the *Generic Connection Framework* in the `javax.microedition.io` package.

Note, however, that no implementation of the GCF is provided by the CLDC configuration. Implementation is left to the profiles.

The CLDC 1.0 specification [3] was released in May 2000, and the CLDC 1.1 specification [4] was released in March 2003. The 1.1 specification is fully backwards compatible with the previous version. Among a few other additions and bug fixes, floating point support was added to version 1.1.

CLDC 1.1 requires the device to have at least 160 kilobyte of memory to store the CLDC implementation, and a minimum of 32 kilobyte available for runtime use (heap etc).

CLDC does not provide any means for persisting data, or communicating with the user.

1.2 MIDP

A profile builds on a configuration to provide additional functionality, like GUI, network implementation and local data storage

MIDP (Mobile Information Device Profile) 1.0 [5] was released in December 2000. As its name suggests, this profile is aimed at mobile devices, like mobile phones or PDAs. It's built on top of the CLDC 1.0 configuration.

Some of the requirements stated by the MIDP 1.0 specification (memory requirements are in addition to those imposed by the underlying configuration):

- Screen size 96*54 pixels, monochrome, square pixels
- Input device, such as keyboard or touch-sensitive screen
- 128 kilobyte memory for storing the MIDP implementation
- 8 kilobyte memory for local data storage
- 32 kilobyte memory for runtime use (heap, etc)
- Two way, limited bandwidth networking (not permanently on-line)

MIDP 2.0 [6] was released in November 2002. It has a number of new features compared to the previous version, and is backwards compatible to make sure MIDlets built on the 1.0 spec will run under a MIDP 2.0 implementation.

A few changes in the hardware requirements are worth mentioning:

- 256 kilobyte memory for storing the MIDP implementation
- 128 kilobyte memory for runtime use (heap, etc)
- The ability to play tones

Some of the new features in MIDP 2.0 are secure networking (https support), multimedia support,more GUI control, a Game API, security mechanisms like code signing and permissions and more [8].

1.3 Optional packages

If all you had to play with was the core MIDP 2.0 APIs, you'd still be quite limited in terms of what functionality your MIDlets has access to. This is why the device vendors are free to include additional APIs that sit on top of MIDP and CLDC. Some of the more important ones are:

- JSR-75, FileConnection and PIM
- JSR-82, Bluetooth API
- JSR-135, Mobile Media API
- JSR-120, Mobile Messaging API

In the rest of this article we'll get some hands-on experience developing MIDlets.

2 Programming MIDlets

2.1 MIDlets and MIDlet suites

As you've probably noticed, programs written for the MIDP profile are called *MIDlets*. A group of related MIDlets can be deployed in the same package. This package is referred to as a MIDlet *suite*. One of the benefits of a MIDlet suite is that the MIDlets constituting the suite can share application data. The synchronization mechanism of Java also spans the MIDlets in a suite. The suite is installed (and removed) on the device as one unit.

The examples provided in this article will deal only with stand-alone MIDlets.

2.2 Using an emulator

It would be a very time consuming process if you had to upload your MIDlet to your device to test every little change in your code. Fortunately, there are emulators implementing the J2ME configurations and profiles that lets you run the MIDlet on your workstation. Sun Microsystems provides a reference implementation of J2ME called the Java Wireless Toolkit (JWT). This includes an emulator for a standard color phone. JWT supports additional APIs, such as Bluetooth Apia's [11] and Web Services [10].

Each device manufacturer also provides development toolkits corresponding to the implementation for their devices. Check out the manufacturer websites for more information.

There is also a good chance your favourite Java IDE is, or can be, J2ME-enabled.

2.3 MIDlet anatomy

All MIDlets inherit from the abstract `javax.microedition.midlet.MIDlet` class. As the name suggests, the MIDlet has a few things in common with an Applet. This includes being downloadable from a remote location, and that it is run in a secure environment, where it has very limitet means to compromise the device it is running on. As you start developing and testing MIDlets, you'll see that the user has to verify a lot of the activities initiated by the MIDlet. This is especially true for all sorts of i/o operations.

The following abstract methods must be implemented by your MIDlet:

```
protected void startApp() {}
protected void pauseApp() {}
protected void destroyApp(boolean unconditional) {}
```

`startApp()` is called when the MIDlet has been constructed by the VM, and is signaled to start. Most of your initialization will go here. `startApp()` is also called by the VM after your MIDlet has been paused. This means that you have to be careful about how you perform your initialization, to avoid allocating resources multiple times.

You can also perform certain initialization in the MIDlet constructor, but you have to do all your GUI initialization in `startApp()`. The MIDP specification

does not guarantee that the GUI (more specifically, the `Display` is available before `startApp()` is called.

When the MIDlet is required to pause (for example if the device is a mobile phone, and the phone rings), `pauseApp()` is called. You should release the resources you don't need when the MIDlet is in the paused state. If the MIDlet itself wants to be paused (for example, the user pauses a game), it should first call `pauseApp()`, followed by a call to `notifyPaused()`. It is the latter call that signals the VM that the MIDlet wants to be paused. The VM will not call `pauseApp()` for you, it assumes it was called before `notifyPaused()`.

Finally, the `destroyApp(boolean unconditional)` method is called by the VM when it wants to terminate the MIDlet. The flag `unconditional` indicates if the MIDlet is required to terminate, or if it can signal (by throwing an exception) that it can't be terminated at this time. All resources allocated by the MIDlet should be released in a proper fashion. If the MIDlet itself wants to be terminated, it should first call `destroyApp()`, followed by a call to `notifyDestroyed()` to signal to the VM that it wants to be terminated. Note that the VM assumes that `destroyApp()` has already been performed before `notifyDestroyed()` was called.

2.4 Example MIDlet

Here's the most basic MIDlet you can dream up. It starts, prints a message, and exits. Note that the output will not be visible if you run this MIDlet on your phone, so you should run this in an emulator, which will have some way to show you what's written to standard out.

The purpose of this MIDlet is to demonstrate the life cycle of the MIDlet.

```
public class TheAnswer extends MIDlet {

  public TheAnswer() {
    System.out.println("Constructor");
  }

  public void startApp() {
    System.out.println("The answer is forty-two");
    destroyApp(true);
    notifyDestroyed();
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
    System.out.println("Destroying");
  }
}
```

2.5 User interface

The GUI components available to a MIDlet can all be found in the `javax.microedition.lcdui` package. A complete run-through is out of the scope of this article, so let's look at the basics.

`Display` is a class representing the MIDlets own, dedicated logical display. The MIDlet fetches a reference to its `Display` by calling the static method `Display.getDisplay(MIDlet MIDlet`. A `Display` has only one active `Displayable` at any given time. A `Displayable` is an object that can be displayed, e.g. a `Form`. A `Form` acts as a container for other GUI components, such as `TextField`s and `ChoiceGroup`s, which are subclasses of `Item`.

Let's look at an example:

```
public class GUIDemo extends MIDlet {
  private boolean started = false;
  private Display d;
  private Form f;
  private TextField tfAlbum;
  private ChoiceGroup cgCategory;

  public void startApp() {
    if(!started) {

      d = Display.getDisplay(this);
      f = new Form("GUIDemo");
      tfAlbum = new TextField("Name", null, 40, TextField.ANY);
      cgCategory = new ChoiceGroup("Category", Choice.MULTIPLE);
      cgCategory.append("Jazz", null);
      cgCategory.append("Soul", null);
      cgCategory.append("Blues", null);

      f.append(tfAlbum);
      f.append(cgCategory);

      d.setCurrent(f);
      started = true;
    }
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
  }
}
```

First, note the use of a flag (`started`) to detect if this is the first time `startApp` is called or not. The `Display` is fetched, and a `Form` is created. Then a `TextField` and a `ChoiceGroup` is created and added to the `Form`. Finally, the newly created `Form` is set as current.

This is probably a good time to have a look at the MIDP API documentation. All we've managed so far is to create a user interface, next we'll look at making it respond to the user's input.

2.6 Commands and command listeners

`Command` represents a (possible) user action. We'll create an exit `Command` in the application above, and add it to our `Form`. In order to be notified if the user initiates the exit `Command`, a class implementing the `CommandListener` interface must be registered with the `Displayable` containing the `Command`. In this example, we'll let the MIDlet itself implement `CommandListener`.

Note that the `commandAction(...)` method should return as quickly as possible, which means that any heavy processing or blocking i/o operation should be performed in separate threads.

```
public class GUIDemo extends MIDlet
       implements CommandListener {
  private boolean started = false;
  private Display d;
  private Form f;
  private TextField tfAlbum;
  private ChoiceGroup cgCategory;
  private final static Command cmdExit =
          new Command("Exit", Command.EXIT, 1);

  public void startApp() {
    if(!started) {

      d = Display.getDisplay(this);
      f = new Form("GUIDemo");
      tfAlbum = new TextField("Name", null, 40, TextField.ANY);
      cgCategory = new ChoiceGroup("Category", Choice.MULTIPLE);
      cgCategory.append("Jazz", null);
      cgCategory.append("Soul", null);
      cgCategory.append("Blues", null);

      f.append(tfAlbum);
      f.append(cgCategory);
      f.addCommand(cmdExit);
      f.setCommandListener(this);

      d.setCurrent(f);
    }
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
  }
```

```
  public void commandAction(Command c, Displayable d) {

    if(c == cmdExit) {

      destroyApp(true);
      notifyDestroyed();
    }

  }
}
```

3 Networking

Next, we'll look at a simple networking example. using a `HttpConnection` to connect to a web server and fetch an XML document from which we'll extract a few bits of information.

To do this, we'll make use of a few classes in the `java.io` and `javax.microedition.io` packages. In J2SE, `java.io` consists of a rather large number of classes and interfaces. To base J2ME i/o on this extensive framework was considered not feasible by the CLDC designers, and so a new framework, the *Generic Connection Framework* was developed. This framework is found in `javax.microedition.io`. A few classes from `java.io` is included in CLDC, including `InputStream`, `OutputStream` and readers and writers for these.

The example fetches an XML file from web server, and gets the first 1024 characters of the document. All the networking takes place in a method called `getLiveWeather()`:

```
protected String getLiveWeather() {

  char[] chars = new char[1024];
  HttpConnection conn = null;

  try {
    conn = (HttpConnection) Connector.open(WEATHER_URL);
    if(conn.getResponseCode() == HttpConnection.HTTP_OK) {

      InputStream is = conn.openInputStream();
      InputStreamReader isr =
        new InputStreamReader(is, "iso-8859-1");
      isr.read(chars, 0, chars.length);
      isr.close();
      isr = null;
      is.close();
    }
  }
  catch(IOException ignored) {}

  return new String(chars);
}
```

First, a buffer is created to hold the characters we're about to read from the web server. It's important to keep in mind the resource constraints that applies to a J2ME device, so be careful when allocating buffers like this. The static method `Connector.open()` is called with a URL as the only parameter. `Connector` looks at the protocol part of the URL (`http` in our case), and opens and returns a `HttpConnection`. `HttpConnection` has a number of useful methods for dealing properly with web server communication (taking care of redirects etc). In J2SE things like redirection logic etc is handled for you. However, this is not the case in the more light-weight GCF. In this example we're just checking that the server responded with a success code.

Given that we have a successful connection, we open an `InputStream`, and build a `InputStreamReader` on top of the stream. The encoding parameter is supplied when creating the reader to avoid `ArrayIndexOutOfBounds` problems when running an emulator where UTF-8 is the standard encoding.

The next step is to fill the buffer with characters, and to return a new `String` built from the characters in the buffer.

Please note that error handling and robustness must be improved (to say the least) for a real life application.

4 Bluetooth

The Java APIs for Bluetooth Wireless Technology [11] are not part of the MIDP specification. It comes as an optional package that builds on MIDP and CLDC to provide access to an existing Bluetooth stack on the device. It is up to the device manufacturer to decide if optional packages are to be included. It should be safe to assume that most MIDP 2.0 devices with a Bluetooth stack also comes with this optional package.

The Bluetooth APIs are quite simple, and consists of two packages: `javax.bluetooth` and `javax.obex`. We'll look at the former package, and see what it takes to discover nearby devices, and connect to one using the serial port profile.

First, the local Bluetooth device must be located, and then its `DiscoveryAgent` can be used to discover nearby devices. In Bluetooth terminology, this process is called *inquiry*. The inquiry process can take some time. Fortunately, it takes place in the background, and a `DisoveryListener` is notified when a device is found, and when the inquiry process is completed. Here's a simple example:

```
public class BTDemo implements DiscoveryListener {
  private String btAddress;

  public void startInquiry() {

    try {
      LocalDevice dev = LocalDevice.getLocalDevice();
      dev.getDiscoveryAgent().startInquiry(DiscoveryAgent.GIAC, this);
    }
    catch (BluetoothStateException e) {
      // Handle exception
    }
  }
```

```
  ...

}
```

The second parameter to `startInquiry()` is a reference to a `DiscoveryListener`. In our example, the `BTDemo` class itself implements this interface, and its `DeviceDiscovered` method looks like this:

```
public void deviceDiscovered(javax.bluetooth.RemoteDevice remoteDevice,
                             javax.bluetooth.DeviceClass deviceClass) {
  try {
    String btName = remoteDevice.getFriendlyName(false);
    btAddress = remoteDevice.getBluetoothAddress();

    // Use name and address
  }
  catch(IOException e) {
    //Handle exception
  }
}
```

Finally, we'll connect to the newly found device, and read from it using a `StreamConnection`.

```
public String connectAndReadLine() {

  if(btAddress == null) {
    return null;
  }

  StringBuffer buffer = new StringBuffer();

  try {
    int c;
    conn = (StreamConnection) Connector.open("btspp://" + btAddress + ":1");
    is = conn.openInputStream();

    while((c = is.read()) != '\n') {
      buffer.append((char) c);
    }
  } catch(IOException e) {
    //Handle exception
  } finally {
    try {
      if(is != null) {
        is.close();
        is = null;
      }
      if(conn != null) {
      conn.close();
```

```
    }
  } catch(IOException ignored) {}

 return buffer.toString();
}
```

The connection to the device is opened using `Connector.open()`, just like when we connected to a web server. The important difference is, of course, the URL parameter. `btspp` is the Bluetooth Serial Port Profile, which lets us communicate with the remote device as if it was a serial port. The `:1` part after the remote device address tells what service channel to use.

After we've successfully opened the connection, one character after the other is read from the stream, until a newline character is encountered. After the connections are closed, the line we read is returned from the method.

Remember that this is just one example of what and how you can use Bluetooth, so take a look at the JSR-82 documentation to find out more.

5 Time to explore

This article has provided some very simple examples, and scratched the surface of some aspects of J2ME development. Here are a couple of ideas for you to explore, if you want to build on these examples:

- Use the Record Management System to store locations and location codes for use in the GetWeather MIDlet.
- Extend this to also the weather report(s) in a `RecordStore`
- Add more sense to the device discovery in the Bluetooth example. Let the user select the remote device to communicate with.
- Create a Bluetooth server which will give nearby phones the current weather from a given location. It's your choice if you want to let the server connect to a web server or to get a stored weather report from the RMS.

References

[1] Kim Topley, *J2ME In a nutshell*, O'Reilly & Assosciates, 1st Edition, 2002

[2] Bluetooth Special Interest Group, http://www.bluetooth.org/

[3] Sun Microsystems, JSR-30, *J2ME(TM) Connected Limited Device Configuration (CLDC) ("Specification")*, Version 1.0a, 2000

[4] Sun Microsystems, JSR-139, *Connected Limited Device Configuration (CLDC) Specification*, Version 1.1, 2003

[5] Sun Microsystems, JSR-37, *Mobile Information Device Profile Specification*, Version 1.0a, 2000

[6] Sun Microsystems, JSR-118, *Mobile Information Device Profile Specification*, Version 2.0, 2002

[7] Java Community Process, http://www.jcp.org/

[8] Jonathan Knudsen, *What's new in MIDP 2.0*, http://developers.sun.com/techtopics/mobility/midp/articles/midp20/

[9] Sun Java Wireless Toolkit, http://java.sun.com/products/sjwtoolkit/

[10] Sun Microsystems, *Web Services Specification*, Version 1.0, 2003

[11] Sun Microsystems, *JavaTM APIs for BluetoothTM Wireless Technology*, Version 1.0a, 2002