keyboard.

the specification document available at JSR 000075 home.

15.4 PROGRAMMING FOR CLDC

Covering the entire gamut of profiles mentioned above is beyond the scope of this book. We will take one profile from the CLDC group, the MIDP. MIDP has been chosen as it is the most widely used and has also created a good amount of noise with the release of its version 2.0. Here unless specifically referred to as introduced by MIDP 2.0 they are applicable to both the versions. The reader, however, is encouraged to study the vendor specific features too as they allow the developer to use the support of the underlying native environment.

A MIDP application is called a MIDlet. It is a take-off on the traditional applet, as the two are similar in some ways. We now look at the MIDlet Model.

15.4.1 The MIDlet Model

A typical MIDP application or MIDlet sits atop the MIDP which in turn requires the services of the CLDC and the VM below it. Finally it is the device hardware that executes instructions on behalf of the software layers above. Figure 15.5 shows a top down view of a MIDP application.

Like an applet a MIDlet needs an execution environment. The browser's equivalent in the MIDP world is called Application Management Software or AMS. It is a device resident software (normally provided by the device vendor) and all MIDlets run within the context of an AMS. This is also required because the handset needs to respond to events outside the MIDlet scope; for example, a call might need to be answered while reading an e-mail. All MIDlets are registered with the AMS during installation. A set of MIDlets can be grouped together into a MIDletsuite. Other than managing the MIDlet, the AMS is also responsible for Application Provisioning and Application Removal.

15.4.2 Provisioning

Provisioning is the process of application discovery, download and installation. PDAs allowed this by downloading the applications on to a PC and then using a serial cable to transfer it to the device. The solution defeats the very purpose of mobility.

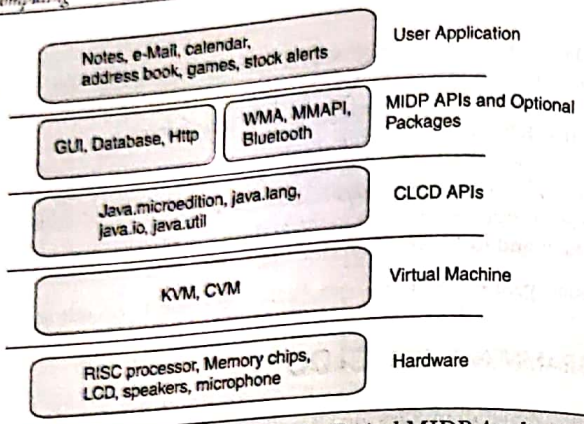

Figure 15.5 A Top Down View of a Typical MIDP Application

Provisioning includes

- **Search:** Which can be performed by the user manually entering the URL where the application is hosted or using a device resident browser step 1 in Figure 15.6.

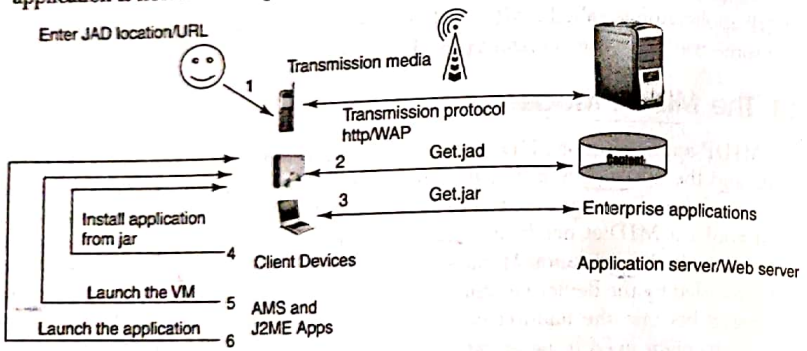

Figure 15.6 Provisioning a MIDP Application

- **Retrieve:** The descriptor file, which includes the application details, checks for version, and compatibility issues, is retrieved (step 2).
- **Download:** The appropriate jar file is downloaded. All middle suites are packaged into a jar file (step 3).
- **Install:** Once the download is completed the AMS is called to install (step 4).
- Finally the VM is launched (step 5) and the
- **Applications** are launched (step 6).

The process includes appropriate user interactions, including payment authorizations. Figure 15.6 depicts the entire process.

MIDP 2.0 has introduced a more interesting and unique capability in the form of a push registry. (MIDP 1.0 specifications allowed only pull applications, i.e., the applications had to initiate the transaction.) These features are discussed further later.

15.4.3 The MIDlet Lifecycle

As shown in Figure 15.7, a MIDlet has three states. A MIDlet class extends javax.microedition.midlet. MIDlet defines the corresponding life-cycle notification methods. These lifecycle methods allow the AMS to notify and request MIDlet state changes.

- Applications are launched either by a user selection or in response to an external event from the push registry. On being activated by the AMS, the MIDlet is constructed but is still inactive. This is the reason why resource allocation is not advisable in the constructor. Now the MIDlet is in a *paused* state.
- Once constructed, the AMS initializes and activates the MIDlet by invoking its startApp() method. The MIDlet now changes to active state. If the initialization fails a javax.microedition.midlet.MIDletStateChangeException is thrown and the state is changed to *destroyed.*
- A transition from *active* state to the paused state is initiated by the AMS by calling the MIDlet's pauseApp() method or by the MIDlet itself through the MIDlet context. In this state a MIDlet should release all its resources.
- A MIDlet can be *destroyed* from either *active* or *paused* state, If destruction is initiated by the AMS, the MIDlet's destroyApp( ) method is invoked with a boolean parameter true/false for optional or forced destruction. The optional destruction can result in a MIDletStateChangeException. The possible state changes and the transitions are shown in Figure 15.7.

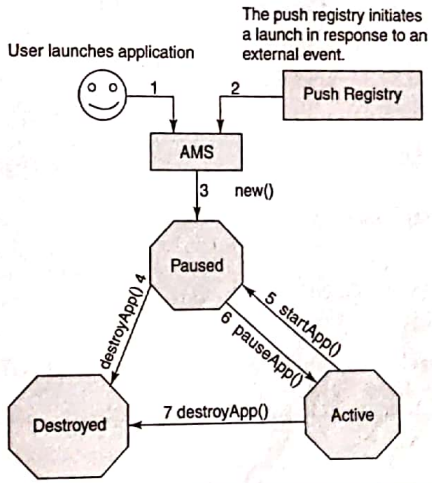

Figure 15.7 MIDlet Lifecycle

Briefly then the three states of a MIDlet are:

- *paused:* The MIDlet is constructed but inactive; transition occurs by a call to pauseApp( ).
- *active:* The MIDlet is active and can process requests; transition occurs by a call to startApp( ).
- *destroyed:* The MIDlet has been destroyed and is ready for garbage collection; transition occurs by a call to destroyApp( ).

Apart from the lifecycle methods javax.microedition.midlet.MIDlet also defines some MIDlet Context methods. These are as follows.

- getAppProperty( ) which retrieves properties from the *application descriptor*. Properties are name-value pairs in the JAD file. The contents of a sample JAD and manifest files are shown in Figure 15.7.
- resumeRequest( ) is a request to the AMS to reactivate the MIDlet. However, it is the prerogative of the AMS to decide if and when to reactivate the MIDlet. Reactivation makes a call to the MIDlet's startApp ( ) method.
- notifyPaused( ) is an alert to the AMS that the MIDlet is transitioning to a paused state.
- notifyDestroyed( ) informs the AMS that the MIDlet will now destroy itself.

Sample JAD file contents

MIDlet-1: GUI, GUI.png, FirstMIDlet
MIDlet-2: SimpleTextBox, SimpleTextBoxMIDlet
MIDlet-3: TextBoxWithCommandListener,
TextBoxWithCommandListenerMIDlet
MIDlet-4: CompleteTextBox, CompleteTextBoxMIDlet
MIDlet-5: SimpleList, SimpleListMIDlet
MIDlet-6: CompleteList, CompleteListMIDlet
MIDlet-Jar-Size: 5698
MIDlet-Jar-URL: GUI.jar
MIDlet-Name: GUI
MIDlet-Vendor: Sun Microsystems
MIDlet-Version: 1.0
MicroEdition-Configuration: CLDC-1.0
MicroEdition-Profile: MIDP-2.0

JAD file is a text file that lists important information about a set of MIDlets packaged together into a single JAR file (*a MIDlet suite*)

Sample MF file contents

MIDlet-1: GUI, GUI.png, FirstMIDlet
MIDlet-2: SimpleTextBox, SimpleTextBoxMIDlet
MIDlet-3: TextBoxWithCommandListener
TextBoxWithCommandListenerMIDlet
MIDlet-4: CompleteTextBox, CompleteTextBoxMIDlet
MIDlet-5: SimpleList, SimpleListMIDlet
MIDlet-6: CompleteList, CompleteListMIDlet
MIDlet-Name: GUI
MIDlet-Vendor: Sun Microsystems
MIDlet-Version: 1.0
MicroEdition-Configuration: CLDC-1.0
MicroEdition-Profile: MIDP-2.0

The *manifest* is the standard JAR manifest packaged with the MIDlet suite.

15.4.4 First MIDLet

The code for our FirstMIDlet would look as below:

```
import javax.microedition.MIDlet.*;
import javax.microedition.lcdui.*;
/**
* @author ryavagal
* @version
*/
public class FirstMIDlet extends MIDlet
{
   private Display display;

   public void startApp( ) throws MIDletStateChangeException
{
         if( display == null ){
            init( ); // one-time initialization
      }
}
public void pauseApp( ) {
   }
public void destroyApp(boolean unconditional) throws
MIDletStateChangeException {
      exit( );
   }
```

```
    private void init( ){
        display = Display.getDisplay( this );
            // initialization stuff goes here
    }
        public void exit( ){
            // It is a good practice to release all
resources and cleanup
            notifyDestroyed( );
        }
}
```

But before you can see this in action you will need to set up your development environment. The easiest way to set up the development environment is to visit http://java.sun.com/j2me/index.jsp where detailed download and installation instructions are given. Integration with the IDE of our choice will involve further investigation on our part. For our purpose we will use the WTK2.0 running on a Windows platform. Details are excluded here as it will only be duplication of content from the Sun's site.

Once we have successfully installed the WTK we can start it from the start menu. This will look like as shown in Figure 15.8. Click on "Open Project" to view any of the sample programs. To run the applications click "Run".

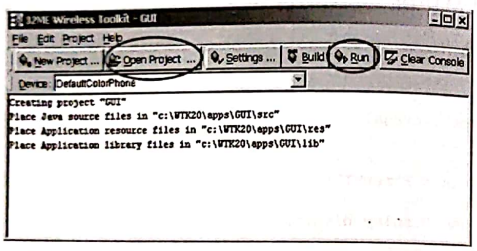

Figure 15.8 Running Samples from WTK

15.4.5 Creating a New Application

To create our own application we need to do the following.

1. Click on "New Project".
2. Enter appropriate names for the application and the application class and click "Create Project". The resulting window is shown in Figure 15.9.
3. By default WTK will create the folder Structure under the Apps folder of the WTK root (Fig. 15.10).

4. Use any editor, (even Notepad will do) to enter the program above and save it under the source folder of the application we created in step 2 .
5. Go back to the WTK and click "Build". If everything is OK we will get a Build complete message (Fig. 15.11).

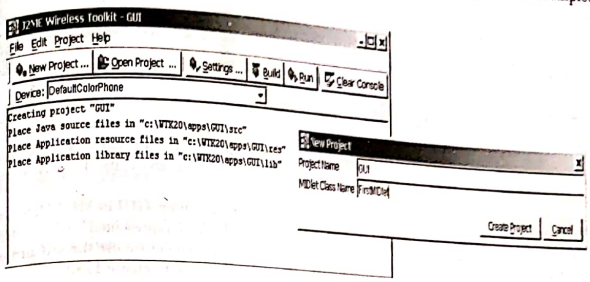

Figure 15.9 Creating a New Application

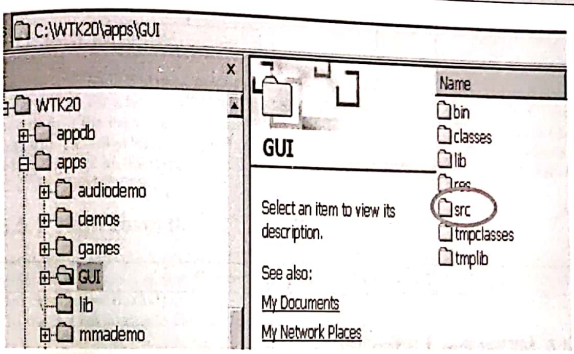

Figure 15.10 Application Project Directory Structure

6. Clicking on run will launch the emulator which lists the Midlet currently registered with the WTK AMS. For steps 4, 5 and 6 refer to Figure 15.11.

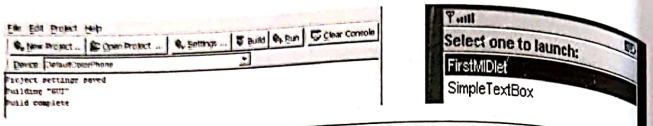

Figure 15.11 Building the First MIDlet

But selecting the FirstMidlet and clicking Launch does nothing! That is because we have not done anything yet. If we take a closer look at the code above there is variable called Display. What is it? And what does it do? And the lcdui package?

We guessed right. We will be using it to display elements on the screen. GUI in MIDP has two core concepts, the *Display* and *Displayable*. In short the MIDP's display is represented by the *Display* class. All displayable elements are called *Displayable*. To show an element we use the setCurrent method of the *Display* class. The lifecycle of GUI interactions is shown in Figure 15.12.

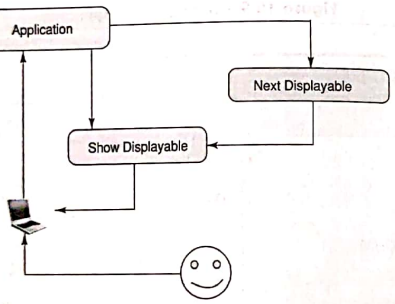

Figure 15.12 Application Control Flow

15.4.6 MIDlet Event Handling

User interactions generate events. These could be:

- Screen inputs.
- Item state change.
- Handset data update.

MIDP event handling mechanism is based on a listener model. It provides interfaces for each of the events mentioned above. These interfaces implement callback methods, which in turn invocate application-defined methods. These methods perform the desired functions in response to events. The three interfaces provided are: ItemStateListener, CommandListener, and RecordListener. Let us look at each in some detail.

Command Listener

The CommandListener as the name implies is responsible for notifying the MIDlet of any commands or events generated by the user. Objects extending it, implement the commandAction method. This method takes two parameters—a Command object and a displayable (Command c, Displayable d). This method implements the functionality that needs to be executed in response to the command event on the associated Displayable.

Displayable.setCommandListener (CommandListener L) sets the listener L to a Displayable.

Item State Listener

An ItemStateListener informs the MIDlet of changes in the state of an interactive item. It calls the itemStateChanged(Item I) method in response to an internal state change.

- Form.setItemStateListener(ItemStateListener L)sets the item state listener L for the given displayable.

Record Listener

RecordListener is related to database events which are discussed in the later section on RMS.

15.5 GUI IN MIDP

As shown in Figure 15.13 the "*Displayable*" has two main subclasses Screen and Canvas. The screen is a super class for a set of predefined UI elements. The predefined UIs are called High Level UI elements and the canvas elements are called Low Level elements. It is always preferable to use these as they involve less coding and are more portable. The Canvas allows the developer to have low level control on the screen. Games normally use the Canvas. Figure 15.13 shows the subclasses of the "*Displayable*".

15.5.1 High Level UI

We will begin with the High Level UIs and then proceed to the Low Level UIs. Let us begin by adding some of the displayables to our applications. Copy the FirstMidlet into a new program called SimpleTextBoxMIDlet.

To the init method, add the following code:

```
Display display = Display.getDisplay(this);
TextBox text = getTextBox( );
display.setCurrent(text);
```

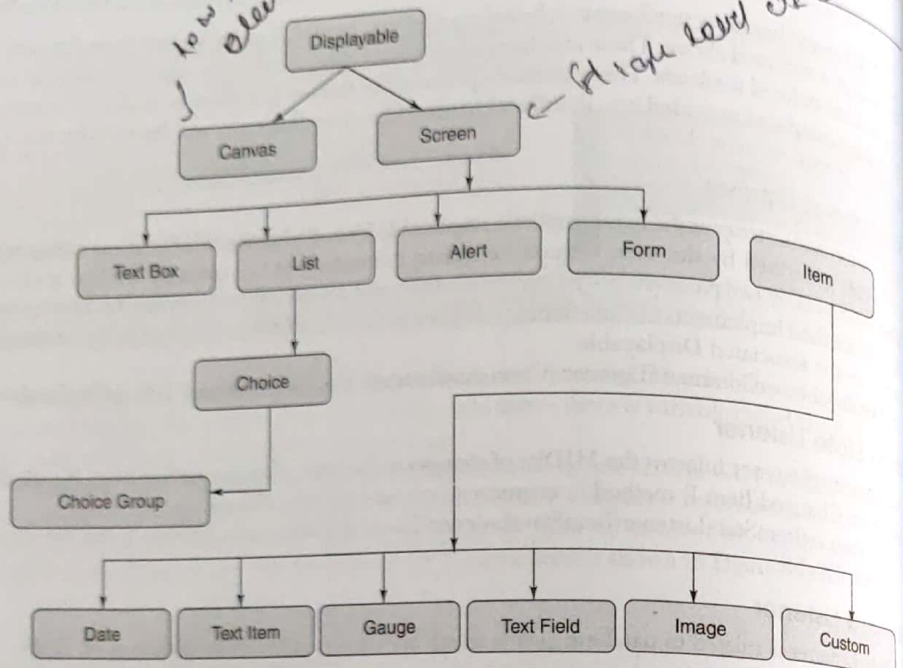

Figure 15.13 UI Elements in MIDP

Now add this new Method

```
public static TextBox getTextBox( )
{
   TextBox textBox = new TextBox("Hello Midlet", "", 50, 0);
   return textBox;
}
```

Save the file Build and Run. Wait a minute, why can't we see it in the list? We need to add the MIDlet to our suite first. Click on settings. We see the DialogBox showing the currently available MIDlets. Refer to Figure 15.14.

- Click on Add in this dialog to add a new MIDlet to our suite.
- Enter The Name to be Displayed and associate it with the Midlet class.
- Click OK.

Now select Run from the WTK toolbar and the newly created SimpleTextBox should appear (Fig. 15.15).

Select SimpleTextBox and Click Launch. We should see the TextBox (Fig. 15.16).

Lets revisit the code above. We first obtain the applications Display, Create the element we want to use and then set the Current to Display to our Element.

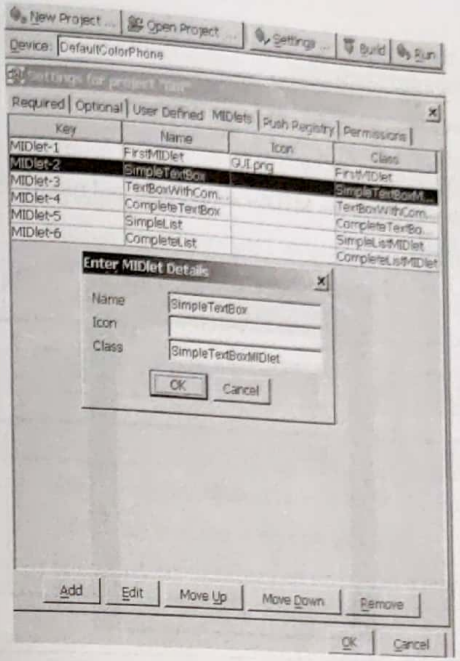

Figure 15.14 Adding a New MIDlet to the Suite

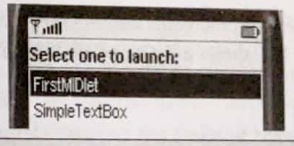

Figure 15.15 New Simple Text Box Appears

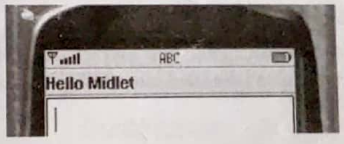

Figure 15.16 Text Box Appears

Forms

A form is similar to its html counterpart. It is used to hold other items and elements. In its simplest form the code for a form might look like

```
display = Display.getDisplay( this );
Form myform = new Form("Hello Form");
display.setCurrent(myform);
```

If we put it in the init method of the MIDlet above build and run we should get the window or form shown in Figure 15.17.

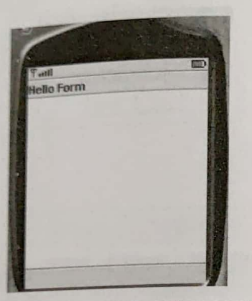

Figure 15.17 Form

Items

To do some useful work we have put some items into the form we created above. MIDP provides a set of predefined items. Table 15.1 shows a listing of all items available in MIDP.

ChoiceGroups

The ChoiceGroup Item allows users to select one or more elements from a group. These groups are similar to the "radio button", "check boxes" and "drop down" elements in the html parlance. AchoiceGroup item contains a simple string and an optional image per member in the group. ChoiceGroups are of three different types. ChoiceGroup and List have a lot of similarity in the options they support. The three types are listed in Table 15.2.

The ChoiceGroup constructor takes a label and a type value. Optionally images and hover text can also be added. Members can also be added after its creation, using the append( ) method.

Table 15.1 Subclasses of Items

Item	*Description*
ChoiceGroup	Allows user to select one or more elements from a group.
DateField	Counterpart of Java date field. Used for date and time values.
Gauge	A bar graph representation used for integer values.
ImageItem	To display an image.
StringItem	Equivalent of html's label widget and used for non-interactive text.
TextField	For text input.
CustomeItem	A user defined item (MIDP2.0).

Table 15.2 Choice Type Constants

Constant	*Value*
EXCLUSIVE	Allows only one element to be selected at a time.
MULTIPLE	Allows the selection of multiple elements.
POPUP	Is a dropdown like construct that allows a single option selection.

```
ChoiceGroup grp = new ChoiceGroup("Select One",Choice.POPUP);
   grp.append("Implicit", null);
   grp.append("Explicit", null);
   grp.append("Multiple", null);
   grp.setLayout(Item.BUTTON);
```

grp.setLayout (Item.BUTTON) allows us to specify the layout, in this case in a button format. To add this to the form use myform.append(grp).

The resulting output is shown in Figure 15.18. The other options of exclusive and multiple are common with another component the List and will be discussed in the section that pertains to LIST.

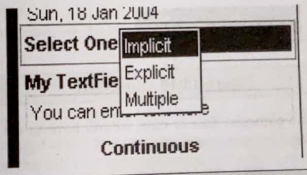

Figure 15.18 ChoiceGroup of Type POPUP

15.6 UI DESIGN ISSUES

- Entering text through the T keypad is not very attractive nor is filling out long forms. UI should be small, simple and easy to use.
- High-level API should be used whereever possible as these are portable across different handsets.
- While using low-level API, it is advisable to remain restricted to elements defined in the Canvas class.
- Device capabilities like screen width, height, and resolution vary from device to device and hence applications should adapt to the Low-level objects accordingly.