Introduction to Mobile Phone Programming in Java Me

(prepared for CS/ECE 707, UW-Madison)

Author: Leszek Wiland and Suman Banerjee

Content

1. Introduction

In this tutorial I will explain the main concepts of developing mobile applications in Java Me. I will start with setting up programming environment in Windows operating system and then proceed with showing Java Me programming model on the example of a number of mobile applications

Java Me is a specification of a subset of Java platform which provides a set of Java APIs for development for small and resource-limited devices. Since this can include many technologically different devices, Java Me is furthermore divided into configurations and profiles which define some certain part of the Java Me APIs which are suitable to be run on a particular device. Mobile Phones have to implement CLDC (Connected Limited Device Configuration) along with MIDP (Mobile Information Device Profile) which both define API which we as programmers can use to build mobile applications. However, both CLDC and MIDP come in different versions suitable for different mobile phones. As of now, Nokia N95 implements the newest version of both CLDC which is 1.1 and MIDP which is 2.0. It is important to mention that a new 3rd version of MIDP is currently under development. Java Virtual Machine for mobile devices has been rewritten and it is not anymore the memory hungry one well known from PC but has been specially optimized for small devices such as mobile phones.

2. Setting up programming environment

There are four main components to be installed in order to start programming in Java for mobile devices. The first one is Java SE (Standard Edition) Development Kit which you might already have installed on your machine; the recommended version is 1.5.0. Next you should download and install Sun Java Wireless Toolkit for CLDC; current version 2.5.2 which can be found at the following link:

http://java.sun.com/products/sjwtoolkit/download.html?feed=JSC

Scroll down to the bottom of the page to find the file. Next install Nokia SymbianOS/S60 SDK for Java. There are multiple versions of this SDK each for a different group of devices. Nokia N95 supports the newest version of the SDK: S60 3rd edition, Feature Pack 2 which can be found under the following link:

http://www.forum.nokia.com/info/sw.nokia.com/id/6e772b17-604b 4081999c31f1f0dc2dbb/S60_Platform_SDKs_for_Symbian_OS_for_Java.html

Look on the right hand side and chose 3rd Edition FP2v1.1 (455 MB) file for downloading. The last thing to install is Eclipse or some other Integrated Development Environment capable of working with Java Me Edition (Java Micro Edition) for example NetBeans with Mobility Pack. In this document I will show how to set up a project with Eclipse so if you don't have Eclipse already installed on your machine you can find it at:

http://archive.eclipse.org/eclipse/downloads/drops/R-3.2.2-200702121330/index.php

An additional Eclipse Me plug-in would be needed for Eclipse to work properly with Java ME and it can be found at the following link:

http://sourceforge.net/project/showfiles.php?group_id=86829

Unpack the zip file and copy all the files in plug-ins and features directories to the corresponding directories in your Eclipse folder. Having done this, we can proceed to configure Eclipse to work properly with Java ME and the phone emulator. Firstly, select Window->Preferences and chose J2ME node and select Device Management. Press import and browse for C:\S60\devices\S60_3rd_MIDP_SDK_FP2_Beta or any other directory you installed the S60 SDK, click refresh and make sure that both S60Emulator and S60Device devices' checkboxes are checked. Press finish and then import again. This time browse for C:\WTK2.5.2 directory, click refresh and make sure that all four devices are selected, then press finish. Back in the main dialog select DefaultColorPhone as the default device. Unfortunately, choosing any other one would make the application fail. Having done that, go to Window->Preferences, expand node Java->Build Path, then For item Source and output folder, select the radio button Folders, and keep the defaults (src and bin). Next expand node Java->Debug and uncheck Suspend execution on uncaught exceptions and Suspend execution on compilation errors, and at the end set Debugger timeout(ms) to 15000 This is enough to start programming for mobile devices and since we have the phone emulator it is easy to test anything we have written. However, in order to move applications easily from computer to a real phone we would need Nokia PC Suite which does that for us. It can be found at:

http://europe.nokia.com/pcsuite

3. Hello World Application

Once all the elements of the programming environment are installed we are ready to create a first Java Me application. Start with opening Eclipse and then select File->New->Project. Expand J2Me node and choose J2ME MIDlet Suite. Press next, give the project a unique name and press next again, also make sure that in the "***Group:***" menu, *Sun Java Wireless Toolkit* is chosen and in "***Device***" menu *DefaultColorPhone* is selected. Press next and select "*Allow output folders for source folders*" and press finish. Once Java Me project is created we need to add a MIDlet class to the project which is the heart of each Java Me application. To do so, right click on the project, select New->Other and choose J2ME MIDlet. This will create a skeleton for Java Me application. Generally, any Java Me program for embedded devices such as Nokia smart phones is called a

MIDlet. There are few differences between Main class of a Standard Java Application and a Main class of a Java Me application. First of all, Java Me Main class does not have main() function, but instead has three empty methods and is derived from MIDlet class (is a subclass of MIDlet class). Each Java Me application has to have a Main class which is derived from MIDlet and contain the following three methods:

1. protected void destroyApp(boolean arg0);
2. protected void pauseApp();
3. protected void startApp();

A MIDlet is simply a backbone of the application and when it is started, Java Virtual Machine would load that Main class and start its startApp() method which in this case acts like the old main() function known from Java SE. The destroyApp(boolean arg0) method is called whenever we want the application to terminate, whereas the pauseApp() is called by the underlying operating system when it needs to free up some extra resources and would like our application to stop executing for some time. This might happen, for example, if there is an incoming call and the operating system needs to free as much resources as possible to deal with this task. Therefore, if our application uses a lot of system resources this would be the place to free them up. However, if we leave this method empty then no resources will be freed by the operating system.

The following simple MIDlet, does not do anything in particular. It is also not capable of any user interaction whatsoever, and therefore cannot be terminated if started on the mobile phone. If run on the computer it can be easily killed by closing the simulator window. The "System.out.println("Hello World");" statement will not have any effect when run on the phone simulator or on the device itself, and therefore cannot be used for printing output.. It would be printed, however, on the Eclipse console and so it is very helpful for debugging.

Code 1

```
public class Main extends MIDlet {

	public Main() {
		// TODO Auto-generated constructor stub
	}

	protected void destroyApp(boolean arg0) {
		// TODO Auto-generated method stub
	}

	protected void pauseApp() {
		// TODO Auto-generated method stub
	}

	protected void startApp() throws MIDletStateChangeException {

		System.out.println("Hey");
	}
}
```

The final step of the development process is to install above Java Me application on mobile device. To do so, we need to generate a jar file which then is copied to the phone. Right click on the project and select J2ME->Create Package (on the very bottom). This will generate one jar and one jad file, both of which are stored under deployed directory in the Java Me project. Once the output files are ready we will install them on the phone device using Nokia PC Suite application. First connect the device to the computer preferably by USB cable and choose PCSuite from the options displayed on the device screen. Then start PC Suite application on your machine and select Install Application item which is on the left bottom of the application window. Browse for the project jar file and by pressing an installation arrow between My Computer and My Phone window copy the jar file to the device where you can finish the installation process.

4. GUI and User Interaction

There are two ways to build a user interface with Java Me. First and the easiest is using high level API which consists of already created components which can be mixed together in almost any fashion. The high level API does also take care of displaying it properly on the screen, scrolling through the menus if needed and basically make our life much easier. All the components have to be eventually attached to an object which is derived from Displayable class to be displayed on the screen. So when using high level API to build a user interface we end up attaching different GUI elements such as TextFields, Buttons, Lists or TextString to an object which is derived from Displayable such as Form, then we set that Displayable to be displayed on the screen, and that is it. This means that we can prepare many different user interfaces or menus and switch between them as easily as it is to set a new Displayable to be displayed on the screen. Once we have the GUI ready we need to take care of user interaction, and this is done by making one of our classes, preferably MIDlet, to implement CommandListener interface which consists of one method:

public void commandAction(Command c, Displayable g);

Code 2

```
public class Main extends MIDlet implements CommandListener{
```

```
	public Main() {
		// TODO Auto-generated constructor stub
	}

	protected void destroyApp(boolean arg0) {
		// TODO Auto-generated method stub
	}

	protected void pauseApp() {
		// TODO Auto-generated method stub
	}

	protected void startApp() throws MIDletStateChangeException {

		System.out.println("Hey");
	}

	public void commandAction(Command c, Displayable g) {
		// TODO Auto-generated method stub
	}
}
```

Any time there is a user generated event the ***commandAction()*** method is called with Command and Displayable arguments passed to it. We can respond to the user input by checking what command has been generated and react accordingly. The above MIDlet implements ***CommandListener*** interface but still cannot process any user input since we have not defined any commands which would let the user to interact with the application. Before proceeding to explain how to do this I will shortly write about the second way of building GUI which is low level API. It lets us directly deal with what is displayed on the screen by defining a ***paint()*** method which is called every time screen has to be refreshed. It also provides a way of capturing any user input, whereas high level API can capture only these event which are directly generated by interacting with buttons displayed on the screen, therefore it misses all the alphanumeric keys. In order to use low level API one of the classes has to be derived from Canvas class and implements the paint method.

protected void paint(Graphics g);

The canvas object defines several other method apart from ***paint()*** which are used mainly for handling low level user interface I will primary focus on ***paint()*** and ***keyPressed()***. The following MIDlet demonstrate use of low level API:

Code 3

```
public class LowLevelAPI extends Canvas{
```

```
        protected void paint(Graphics arg0) {
              // TODO Auto-generated method stub

        }

        protected void keyPressed(int keyCode) {

        }

}
```

Like ***commandAction()*** method for high level API we check the keyCode and depending on the key code take up some specific actions.

So far, we know that there are two types of API available for building GUI and that user input can be handled by either implementing ***commandAction()*** or ***keyPressed()*** method or both of them. Now I will proceed with showing how to build a simple GUI using high level API. As I mentioned before only the elements which are derived from Displayable or Screen can be displayed on the device. Screen has 4 main subclasses each of which can be used to build GUI, and when ready, be displayed on the screen by setting it as a current Screen. These subclasses are: Alert, List, TextBox and Form.

- **Alert:** is a screen that shows data to the user and waits for a certain period of time before proceeding to the next Screen.

- **List**: contains a list of choices and the set of methods to query what option have been selected by the user.

- **TextBox**: allows user to input text.

- **Form**: can contain an arbitrary mixture of items: images, read-only text fields, editable text fields, editable date fields, gauges, choice groups, and custom items each of which can be attached to the form in any fashion.

The implementation handles layout, traversal and scrolling automatically. Apart from these four derived from the Screen classes there is a number of components derived from Item class, which can be attached to a Form and used as a building blocks of the whole GUI.

In order to handle user interface we need to define a set of commands which are then attached to the Displayable. It can be any class which is derived from Displayable, such as: Alert, List, TextBox, Form and also the low level API class: Canvas. Each command is represented by a separate Command object with a name, command type and the priority. Once we have GUI and commands defined, we need to register this Displayable object with a ***CommandListener***, which is done by calling ***setCommandListener()*** method on the Displayable object.

I will show now how to build an application which lets users type a text in one window and prints it in the other one. It has two buttons, one for closing the application and the other one to notify the application that the text should be printed on the screen. I will use the following elements of the high level API: Form, TextField, StringItem and two commands for user interaction.

Code 4

```
public class Main extends MIDlet implements CommandListener{

	Form form = null;
	TextField input = null;
	StringItem output = null;
	Display display = null;

	Command ready = null;
	Command exit = null;

	public Main() {

		// give the form name
		form = new Form("GUI Demo");
		// arguments TextField: name, default text, size in letters, type
		// of input
		input = new TextField("Input", "Write Sth", 10, TextField.ANY);
		// arguments for StringItem: label, name
		output = new StringItem("Output","");
		// arguments for Command: Name, type, priority
		ready = new Command("Display", Command.SCREEN, 1);
		exit = new Command("Exit", Command.EXIT, 1);
		// get display object for this MIDlet
		display = Display.getDisplay(this);
		// add commands to the form
		form.addCommand(ready);
		form.addCommand(exit);
		// add other elements to the form
		form.append(input);
		form.append(output);
		// register the form with Command Listener
		form.setCommandListener(this);
	}

	protected void destroyApp(boolean arg0) {
		notifyDestroyed();

	}

	protected void pauseApp() {
		// TODO Auto-generated method stub
	}

	protected void startApp() throws MIDletStateChangeException {

		display.setCurrent(form);
	}
```

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

                if(c == exit)
                        destroyApp(false);
                else if(c == ready) {
                        output.setText(input.getString());
                }
        }
}
```

The first step is to defined all the building elements of the GUI. This is ***TextField*** for user input, ***StringItem*** for the output, Form to bind all the elements together, and two Commands for user interaction. The next step is to retrieve the Display object for this MIDlet, which is used to switch between Screens displayed on the device. This is done by calling static method of Display class which takes as an argument MIDlet object for which the display should be returned. Once this is done, we add the previously defined command to the Form along with any other items by calling ***addCommand()*** and ***append()*** method on the form respectively. The very last thing to make the GUI work is to register it (in this case form) with the ***CommandListener*** which is accomplished by calling ***setCommandListener()*** method on the form object. The ***startApp()*** method has also been slightly changed. When started, MIDlet would set our GUI to be displayed on the device by calling ***setCurrent(form)*** method on the display object which was retrieved at the beginning. Since the GUI is ready we are able to fill in the ***commandAction()*** method to respond to user generated events. We have two Commands: Exit in which case the ***destroyApp()*** method is called and Ready which retrieves a string from editable text field and prints it out in the non-editable field.

5. Networking

Java Me defines an extremely flexible API for network connections. It is based around a number of Connection interfaces each one for a particular connection type and a Connector class which is a factory for creating new Connection Objects. MIDP specification requires devices which implement it to support at least ***HttpConnection***, however, most of current mobile phones including Nokia N95 support a range of different connection types such as: ***CommConnection***, ***DatagramConnection***, ***SocketConnection***, ***HttpConnection***, ***HttpsConnection*** (secure http) or ***SocketServerConnection***. Majority of these connections allow to write a client application, however by using ***SocketServerConnection*** it is also possible to write a server application for mobile platform. Creating Connection object is done by calling open() method of Connector class. It takes string argument to define the type of Connection to be returned. The basic connection descriptors are:

- **SocketConnection:**

  socket://hostIP:portNo – connects to hostIP:portNo

- **SocketServerConnection:**

  socket://:portNo – listens on port: portNo

- **DatagramConnection:**

  datagram://:port – listens on port: portNo

  datagram://hostIP:portNo – sends datagrams to hostIP:portNo

- **HttpConnection:**

  HTTP URL

I will show a simple networking application which connects to Google server via http connection to download and display Google map on the device screen. Since networking API we use is a blocking API it is important to locate this code in a separate thread in order not to block the whole application so that a user can interact with the application while map is being downloaded.

Code 5

```
public class Main extends MIDlet implements CommandListener{

	Form form = null;
	Image googleImage = null;
	Display display = null;
	Command displayImage = null;
	Command exit = null;

	public Main() {

		// give the form name
		form = new Form("GUI Demo");
		// arguments for Command: Name, type, priority
		displayImage = new Command("Image", Command.SCREEN, 1);
		exit = new Command("Exit", Command.EXIT, 1);
		// get display object for this MIDlet
		display = Display.getDisplay(this);
		// add commands to the form
		form.addCommand(displayImage);
		form.addCommand(exit);
		// register the form with Command Listener
		form.setCommandListener(this);
	}

	protected void destroyApp(boolean arg0) {
		notifyDestroyed();
	}

	protected void pauseApp() {
```

```
	}

	protected void startApp() throws MIDletStateChangeException {
		display.setCurrent(form);
	}

	public void commandAction(Command c, Displayable g) {
		if(c == exit)
			destroyApp(false);
		else if(c == displayImage) {
			Connection con = new Connection(this);
			con.start();
		}
	}
}

public class Connection extends Thread {

	Main parent = null;
	String url = "";

	public Connection(Main parent) {

		this.parent = parent;
	}

	public void run() {

		DataInputStream response = null;
		byte[] receivedImage = null;
		url = "http://maps.google.com/staticmap?center=51.510605,-
		0.130728&format=png32&zoom=8&size=" +
		new Integer(parent.form.getHeight()).toString() + "x" + new
		Integer(parent.form.getWidth()).toString() +
		"&key=ABQIAAAAnfs7bKE82qgb3Zc2YySoBT2yXp_ZAY8_ufC3CFXhHIE1NvwkxSy
		Sz_REpPq-4WZA27OwgbtyR3VcA";

		try {
			HttpConnection c = (HttpConnection) Connector.open(url);
			response = new DataInputStream(c.openInputStream());
			receivedImage = new byte[(int)c.getLength()];
			response.readFully(receivedImage);
			System.out.println("Downloading ok");
		}
		catch(IOException e) {
			System.out.println(e);
			System.out.println("Downloading error");
			Alert alert = new Alert("", "Network error", null,
					AlertType.INFO);
			parent.display.setCurrent(alert, parent.form);
		}
		finally {
			try {
				response.close();
			} catch(IOException e) {
				System.out.println(e);
```

```
                }
            }

            // image in a byte array.
            // transform it in a LCUI Image object

            System.out.println("Start creating image");

            try {
                parent.googleImage = Image.createImage(receivedImage, 0,
                receivedImage.length);
                System.out.println("Image init ok");
            }
            catch(Exception e) {
                System.out.println(e);
                System.out.println("Image init error");
                Alert alert = new Alert("", "Image error", null,
                          AlertType.INFO);
                parent.display.setCurrent(alert, parent.form);
            }

            //display image
            parent.form.deleteAll();
            parent.form.append(parent.googleImage);
        }
}
```

The Main class defines application's user interface but it does not do any networking. Once a user press "Display" command the application creates Connection object and runs it as a separate Thread. Connection object initiate the http connection, downloads the image, displays it on the screen and terminates. It is important to notice that the Connection object does not have any way of displaying content on the device screen since it does not define any graphical user interface. However, the user interface is defined by Main class, therefore parent reference is passed to Connection object which can use the parent user interface to display the image on the screen.