Mount Kenya University

UNIT CODE: BIT4107

UNIT TITLE: MOBILE APPLICATIONS DEVELOPMENT

BIT4107: MOBILE APPLICATIONS DEVELOPMENT

Credit Hours: 3

Contact Hours: 42

Prerequisite: Data Communication and Networks, Object-Oriented Programming

Purpose

To teach students how to develop mobile applications using the Java 2 Platform

Expected Learning Outcomes

By the end of the course unit a learner shall be able to:

i. use the Objective-C and Java languages (and associated frameworks) for creating mobile apps on iOS and Android platforms, respectively .

ii. create effective user interfaces for mobile apps .

iii. harness Internet service in support of mobile apps .

iv. store/retrieve data in support of mobile apps.

Course Content:

Introduction to Mobile Application Programming; Mobile Devices, Java 2 Micro Edition, CLDC, CDC, MIDP, MIDlets; High Level UI, Display, Displayable, Command, Ticker, Screen, Item, Alert, List, TextBox, Form; Low Level UI, Canvas, Graphics; Persistent Storage, Record Stores, Record Enumeration, Record Comparator, Record Filter; Networking, Generic Connection Framework, HTTP Connection, HTTPS Connection, TCP Sockets Server Sockets, Datagrams; Optimizations, Program Execution, JAR Size Networking, Memory Usage; Optional Packages, Overview of optional packages, MMAPI, WMA

Teaching / Learning Methodologies: Lectures and tutorials; group discussion; Demonstration; Individual assignment; Case studies

Instructional Materials and Equipment: Projector; test books; design catalogues; computer laboratory; design software; simulators

Assessment

Examination - 70%; Continuous Assessment Test (CATS) - 20%; Assignments - 10%; Total - 100%

Core text books

Burnette, Ed, Hello, *Android: Introducing Google's Mobile Development Platform*

Recommended Books

Jones, M. and Marsden, *Mobile Interaction Design Scaling the Heights of Education*

Table of Contents

1. Introduction to Mobile and wireless communications

Objectives:

At the end of this topic the students should be able to:

- Understand Mobile communication.
- Understand wireless communication
- Understand the GSM network.
- Understand the G-Networks(2G,3G,4G)

Mobile Communication Technologies

Mobile communication allows transmission of voice and multimedia data via a computer or a mobile device without having connected to any physical or fixed link. **Mobile communication** is evolving day by day and has become a must have for everyone. **Mobile communication** is the exchange of voice and data using a communication infrastructure at the same time regardless of any physical link.

There are so many types of mobile computers, such as laptops, PDAs, PDA phones and other mobility devices were introduced in the mid of 1990s including wearable technology as well. And to use these types of mobility The mobile communication technologies we can count on many mobile technologies available today such as **2G**, **3G**, **4G**, **WiMAX**, **Wibro**, **EDGE, GPRS** and many others.

WiBro (Wireless Broadband) is a wireless broadband Internet *technology* developed by the South Korean telecoms industry.

2G Technology

Second generation (2g) telephone **technology** is based on GSM or in other words global system for mobile communication.

How 2G works and its uses of 2G technology.

2G technologies enabled the various mobile phone networks to provide the services such as text messages, picture messages and MMS (multimedia messages).

2G technology is more efficient. 2G technology holds sufficient security for both the sender and the receiver. All text messages are digitally encrypted. This digital encryption allows for the transfer of data in such a way that only the intended receiver can receive and read it.

Second generation technologies are either time division multiple access (**TDMA**) or code division multiple access (CDMA). TDMA allows for the division of signal into time slots.

CDMA-allocates each user a special code to communicate over a multiplex physical channel, different TDMA technologies include GSM, PDC, iDEN, iS-136.

PDC-(Program delivery control), a system for broadcasting a coded signal at the beginning and end of a television program which can be recognized by a video recorder and used to begin and end recording.

iDEN- Integrated Digital Enhanced Network (iDEN) is a mobile telecommunications technology, developed by Motorola, which provides its users the benefits of a trunked radio and a cellular telephone. ..

CDMA technology is IS-95.

GSM -this enabled the mobile subscribers to use their mobile phone connections in many different countries of the world based on digital signals ,unlike 1G technologies which were used to transfer analogue signals. GSM has enabled the users to make use of the short message services (SMS) to any mobile network at any time.

3G Technology

GSM technology was able to transfer circuit switched data over the network. The use of **3G technology** is also able to transmit packet switch data efficiently at better and increased bandwidth. 3G mobile technologies offer more advanced services to mobile users.

The spectral efficiency of 3G technology is better than **2G technologies**. 3G is also known as IMT-2000.

3G technology characteristics

3G technologies make use of TDMA and CDMA. 3G (Thrid Generation Technology) technologies make use of value added services like mobile television, GPS (global positioning system) and video conferencing.

The basic feature of 3G Technology (Thrid Generation Technology) is fast data transfer rates.

It is expected that 2mbit/sec for stationary users, while 348kbits when moving or traveling. 3G technology is much flexible, because it is able to support the 5 major radio technologies. The radio technologies operate under CDMA, TDMA and FDMA.

CDMA holds for IMT-DS (direct spread), IMT-MC (multi carrier). TDMA accounts for IMT-TC (time code), IMT-SC (single carrier).

FDMA has only one radio interface known as IMT-FC or frequency code. 3G (Thrid Generation Technology) system is compatible to work with the 2G technologies.

There are other many **3G** technologies as W-CDMA, GSM EDGE, UMTS, DECT, WiMax and CDMA 2000.Enhanced data rates for GSM evolution (EDGE) is a backward digital technology, because it can operate with older devices. EDGE allows for faster data transfer than existing GSM. EDGE is a 3G Technology (Thrid Generation Technology); therefore it can be used for packet switched systems. Universal mobile telecommunications systems .UMTS conforms to ITU IMT 2000 standard (International Telecommunication Union). It is complex network and allows for covering radio access, core network and USIM (subscriber identity module). It is a relatively expensive technology for the network operators because it requires new and separate

infrastructure for its coverage. The GSM is the base of this technology. CDMA is also referred to as IMT-MC. this technology is close to 2G technology GSM because it is also backward compatible

WiMax is a 3G Technology (Thrid Generation Technology) and it is referred to as worldwide interoperability for microwave access. It is a wireless technology. It can be operated on the multi point and point modes. it is portable technology. This technology is based on the wireless internet access. This technology removes the need for wires and is capable enough to provide 10mbits/sec. it can connect you to hotspot.

4 G Technology

4G (**Generation Technology)**. **4G Technology** is basically the extension in the **3G technology** with more bandwidth and services offers in the 3G. The implementation and access to the **4G network** is still in it development stage. Some people say that **4G technology** is the future technologies that are mostly in their maturity period. The expectation for the 4G technology is basically the high quality audio/video streaming over end to end Internet Protocol. **WiMAX** or mobile structural design will become progressively more translucent, and therefore the acceptance of several architectures by a particular network operator ever more common.

The technologies that fall in the **4G** categories are UMTS, OFDM, SDR, TD-SCDMA, MIMO and WiMAX to the some extent.

4G Technology offers high data rates that will generate new trends for the market and prospects for established as well as for new telecommunication businesses. **4G networks**, when tied together with mobile phones with in-built higher resolution digital cameras and also High Definition capabilities can facilitate video blogs.

Fourth Generation is the latest technology with high speed transferability of data with security measurements. It is coming with wireless broadband for the instant download.

After successful implementation, **4G technology** is likely to enable ubiquitous computing, that will simultaneously connects to numerous high date speed networks offers faultless handoffs all over the geographical regions.

How 4G Works

The standard of **4G technology**, still not defined as set standard, two technologies are supposed to be the based features of 4G.

- WiMAX
- LTE

ITU promotes the technologies against the defragmentation and incompatibilities in 4G technologies.

WiMAX stands for Worldwide Interoperability of Microwave Access previously worked as fixed wireless facility under the 802.16e band. Now the modified standard 802.16m has been developed with the properties of speed, wide spectrum, and increase ban

Smartphones with Wireless Access are going to be introduced in the market are the model 4G mobiles. These smartphone are equipped with the wireless internet accessibility and no fear of losing connection while travel from one tower to another tower range.

Based on the IP wireless connectivity, it increases the optimization for the internet. It manages the voice through packet-switching instead of circuit switching. Internet connectivity with specific IP not only increases the speed but also reliability of the sending and receiving of data.

During a phone call when caller send the information by connecting to WiMAX network, this information first processed to the internet home and then spread widely. Most of the time this transmission happens very fast problems arise in case of spectrum, bandwidth and data. With 4G the fears of lower bandwidth, narrow spectrum and amount of data send / receive is diminished.

4G Network

4G network or wireless communication is not a new terminology as 1G, 2G & 3G are already in the market and has been used by the people since 1980. All generations have their particular traits to attract the user for its use. 1G is the first complete wireless standard removes the fuss of wiring and per channel 1 user. The lodging of multi user in one channel becomes possible when 2G comes with improved version after a decade (1990) of 1G. 3G revolutionized the concept of

internet. 3G is the 21st generation network and is enjoying its privilege. The successor of 3G is now its way of implementation and is known as 4G. The video facility and speed which is introduced by 3G will be the true essence of 4G. It promises all basic features together with high speed.

Wireless communications

Wireless communication is the transfer of information over a distance without the use of electrical conductors or wires. today the term is used to describe modern wireless connections such as in cellular networks and wireless broadband Internet. It is also used in a general sense to refer to any type of operation that is implemented without the use of wires, such as "wireless remote control".

Wireless communications encompasses various types of fixed, mobile, and portable two way radios, cellular telephones, personal digital assistants (PDAs), and wireless networking.

Common examples of wireless equipment in use today include:

- Cellular Telephones and Pagers
- Global Positioning System (GPS)
- Cordless Computer Peripherals
- Cordless Telephone Sets
- Satellite Television
- Wireless Gaming equipment.

How to achieve wireless Communication:

This may be one-way communication systems such as television or radio broadcasting, or two way communication systems such as machine-to-machine (M2M).

Wireless communication can be via;

i) Radio frequency
ii) Microwave.(i.e. long-range and line-of-sight via antennas or short range communications)
iii) GSM-Global System for Mobile Computation

iv) GPRS-General Packet Radio System.
v) 3G/HSDPA-High Speed Downlink Packet Access
vi) GPS-Global Positioning System.
vii) ZigBee

GSM (Global System for Mobile Communications)

GSM is a global standard for digital cellular communications.GSM uses narrowband Time Division Multiple Access (TDMA) for voice and Short Messaging Service (SMS).

GSM is a cellular network that allow mobile devices to connect to it from cells in the immediate vicinity. The GSM is a circuit-switched system that divides each 200kHz channel into eight 25kHz time-slots. GSM operates in the 900MHz and 1.8GHz bands in Europe and the 1.9GHz and 850MHz bands in the US.

A GSM digitizes and compresses data, then sends it down through a channel with two other streams of user data, each in its own time slot. It operates at either the 900 MHz or 1,800 MHz frequency band.

Why use GSM

The GSM study group aimed to provide the followings through the GSM:

- Improved spectrum efficiency.
- International roaming.
- Low-cost mobile sets and base stations (BSs)
- High-quality speech
- Compatibility with Integrated Services Digital Network (ISDN) and other telephone company services.
- Support for new services.

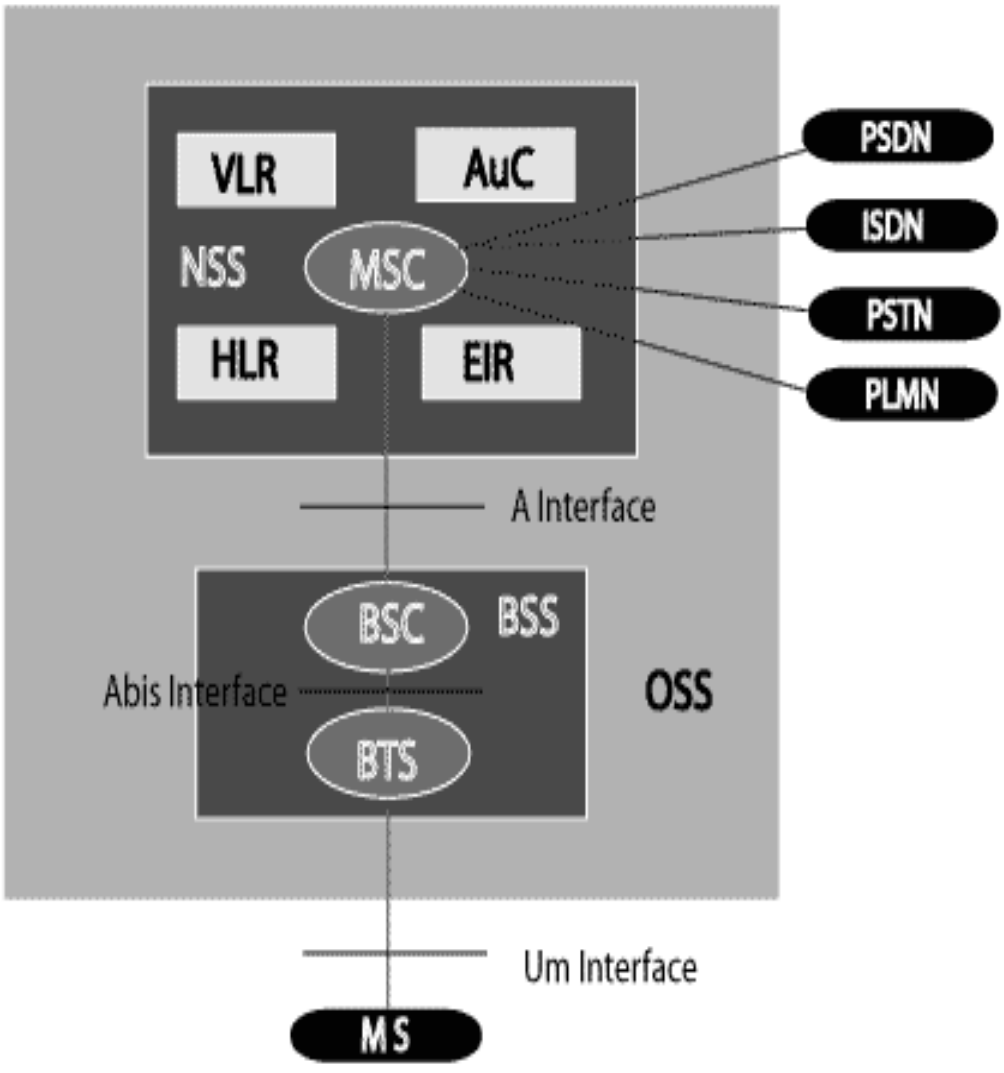
The GSM network has five different cell sizes – macro, micro, pico, femto and umbrella cells. The coverage area of each cell varies according to the implementation environment.

GSM - Architecture

A GSM network consists of several functional entities whose functions and interfaces are defined. The GSM network can be divided into following broad parts.

- The Mobile Station(MS)
- The Base Station Subsystem (BSS)
- The Network Switching Subsystem (NSS)
- The Operation Support Subsystem(OSS)

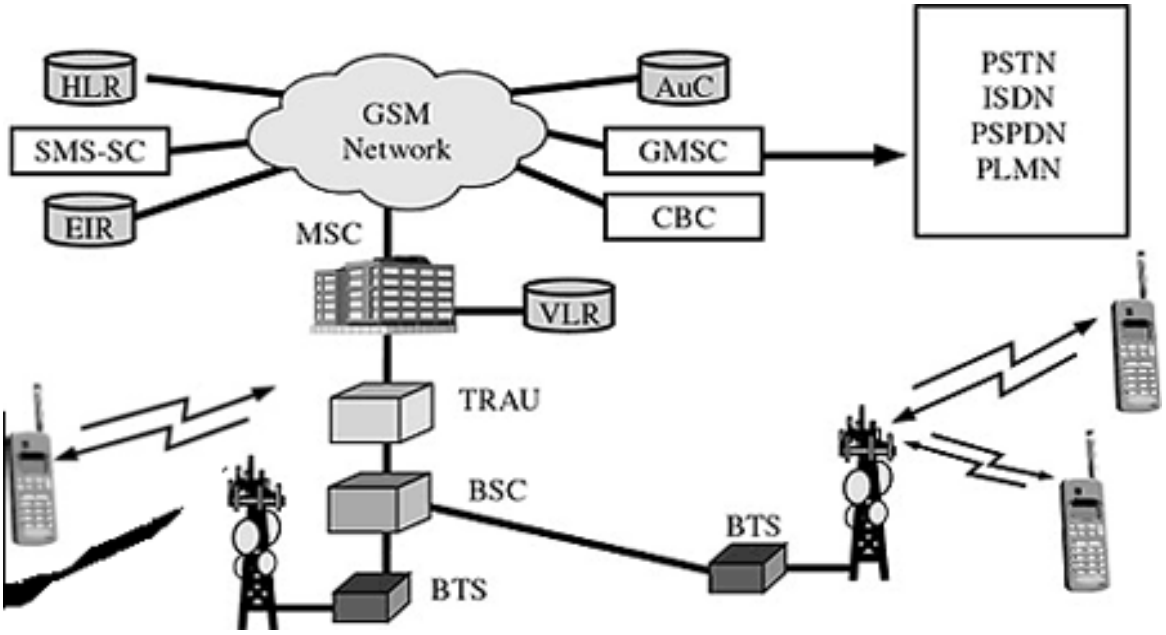
Following is the simple architecture diagram of GSM Network.



The added components of the GSM architecture include the functions of the databases and messaging systems:

- Home Location Register (HLR)
- Visitor Location Register (VLR)
- Equipment Identity Register (EIR)
- Authentication Center (AuC)
- SMS Serving Center (SMS SC)
- Gateway MSC (GMSC)
- Chargeback Center (CBC)
- Transcoder and Adaptation Unit (TRAU)

Following is the diagram of GSM Network alongwith added elements.



The MS and the BSS communicate across the Um interface, also known as the air interface or radio link. The BSS communicates with the Network Service Switching center across the A interface.

GSM network areas:

In a GSM network, the following areas are defined:

- **Cell:** Cell is the basic service area: one BTS covers one cell. Each cell is given a Cell Global Identity (CGI), a number that uniquely identifies the cell.

- **Location Area:** A group of cells form a Location Area. This is the area that is paged when a subscriber gets an incoming call. Each Location Area is assigned a Location Area Identity (LAI). Each Location Area is served by one or more BSCs.
- **MSC/VLR Service Area:** The area covered by one MSC is called the MSC/VLR service area.
- **PLMN(**Public Land Mobile Network**):** The area covered by one network operator is called PLMN. A PLMN can contain one or more MSCs.

Revision Exercise.

1. Define Mobile communication and explain its salient characteristics.
2. Explain wireless communications and how it can be achieved
3. Explain 2G network works.
4. Explain the term cellular network.
5. Discuss 4G network.
6. Explain the characteristics of 3G technology.
7. Explain GSM and it architecture.

2. Java Advance Topics-Graphical User Interface.

Objectives:

At the end of this topic the students should be able to:

- Understand Java basic classes.
- Understand Java GUI.
- Create a simple GUI.

Introduction.

Java Overview

Java includes three different editions J2SE (Java 2 Standard Edition) ,J2EE (Java 2 Enterprise Edition) ,J2ME (Java 2 Micro Edition) .These three editions target for different devices or systems.

The source file is what holds the source code to a program. In Java, the source file can be a text file that contains one or more class definitions.

The Java compiler requires that a source file to have .java filename extension. The source code is converted into bytecode. Bytecode is later translated by the interpreter.

The name of that class should match the name of the file that holds the program.

Keyword class declares that a new class is being defined. **public static void main (String args[]).** Begins the main( ) method. Point at which the program will begin executing.

First Java Program

```
/*
This is a simple Java program name the file Ours.java.
*/
class Ours {
// A Java program begins with a call to main().
public static void main(String args[]) {
System.out.println(“Hello Java programming.");
}
}
```

Another java program that uses the scanner class to capture two inputs from the keyboard and compute the sum of the same.

```
import java.util.Scanner
public class Addition
{
public static void main(String args[])
{
Scanner input = new Scanner(System.in)
int num1,num2,sum;
System.out.print(“Enter value One”);
num1 = input.nextInt();
System.out.print(“Enter value two”);

num2 = input.nextInt();

sum =num1+num2;

System.out.println(“The sum of the two values are:”+sum);
```

}

}

The import in line one uses Java's predefined Scannerclass from package java.util.

Scanner input = new Scanner(System.in);

This is a variable declaration statement that specifies the name (input) and type (Scanner) of a variable that is used in this program.

num= input.nextInt(); Uses scanner object input's nextInt to obtain an integer from the user.

Java object and classes

Class Declaration.
Declaration one

```
class MyClass
{
//field, constructor, and method declarations
}
```

Another Declaration

```
class ourClass extends myClass implements OurInterface
{
//field, constructor, and method declarations
}
```

Means that OurClass is a subclass of OurSuperClass and that it implements the OurInterface interface.

Java constructors

Constructors-used to create objects of a class.

Person ps=new Person("John");

Or Rectangle rectTwo= new Rectangle(50, 100);

Working with the JOptionPane Class.(Message Dialog Box and Input dialog)

You can use the showMessageDialog method in the JOptionPane class.

JOptionPane is one of the many predefined classes in the Java system, which can be reused.

The showMessageDialog Method

JOptionPane.showMessageDialog(null,"Welcome to Java!","Example 1.2", JOptionPane.INFORMATION_MESSAGE);

Incorporate this code inside the main method and execute the application it should be able to produce the following output.

Figure 1. Message Output.

Another example of a message dialog.

JOptionPane.showMessageDialog(null,"Welcome to Java!", “Display Message",JOptionPane. INFORMATION_MESSAGE);

Figure 2: Message Output.

Introduction to Java Graphical User Interface.

In this section, we'll learn how to use in your own program the push buttons, radio buttons, text boxes, and other components found in commercial programs.

User Interface

An interface is a way to interact with something. For example, your television remote control is an interface to your television.

A user interface is the way someone interacts with a program.The simplest user interface consists of two components: a prompt displayed on the screen and the keyboard used to enter information into the program.

The prompt might be some text that tells the user of the program to enter a student ID. The user then enters the student ID into the program using the keyboard. The program then processes the student ID when the ENTER key is pressed.

GUI

GUI (Graphical User Interface). Practically every program used today uses graphics as a way for a user to interact with the program.

GUI is an intuitive and efficient way to collect information from a user and to display information for a user to read. At the heart of a GUI are the standard graphical elements that collectively form the user interface. These elements are commonly recognized as **windows,**

menus, push buttons, labels, text boxes, radio buttons, and other similar GUI objects that you see used in nearly all commercial programs today.

You have two ways in which you can create a GUI for your program.

i. First is by using a message dialog box(discussed previously) and an input dialog box.

   A dialog box is a small window.

   A **message dialog box** is a dialog box that displays a message and an OK push button, which is what you see whenever a warning message is displayed on the screen.

An **input dialog box** is similar to a message dialog box, except the user enters information into an input dialog box.

Both the message dialog box and the input dialog box are displayed by calling one method.

ii. The second way to create a GUI for your program is to use radio buttons, push buttons, and other GUI objects that you see in most programs.

A Simple GUI

The easiest way to create a GUI for your program is to use a message dialog box and an input dialog box. A message dialog box is used to display a message on the screen, and an input dialog box prompts the user to enter information that is returned to your program for processing.

Message Dialog Box

You create a message dialog box by calling the **showMessageDialog()** method, which is defined in the JOptionPane class contained in the **javax.swing** package.

This means you'll need to import the **javax.swing** package at the top of your program in order to call the showMessageDialog() method.

The showMessageDialog() method requires four arguments.

- The **first argument** is a reference to the parent that calls the showMessageDialog() method.
- The **second argument** is the message you want displayed on the screen.
- The **third argument** is the caption that appears in the title bar of the dialog box.
- The **fourth argument** is a constant that states the kind of message dialog box you want displayed.

The following are the most commonly used message constants. Each one displays an icon within the message dialog box that corresponds to the kind of message being displayed.

Message dialog type	Description
ERROR_MESSAGE	A dialog that indicates an error to the user.
INFORMATION_MESSAGE	A dialog with an informational message to the user.

WARNING_MESSAGE	A dialog warning the user of a potential problem.
QUESTION_MESSAGE	A dialog that poses a question to the user. This dialog normally requires a response, such as clicking a Yes or a No button.
PLAIN_MESSAGE	A dialog that contains a message, but no icon

The **showMessageDialog() method** also displays an OK push button that the user selects to acknowledge the message. The push button closes the dialog box when selected.

Here is how you call the **showMessageDialog() method**:

JOptionPane.showMessageDialog(null,"Message","Window Caption", JOptionPane.PLAIN_MESSAGE);

Input Dialog Box.

An input dialog box is created by calling the **showInputDialog()** method, which is also defined in the **JOptionPane** class. The **showInputDialog()** method requires one argument, which is a message prompting the user to enter information into the dialog box.

The showInputDialog() method displays the message and a text box, which is where the user enters information. In addition, the OK and Cancel push button are displayed. Selecting OK causes the showInputDialog() method to return the information that the user entered into the text box. Selecting either push button closes the dialog box.

Information that the user enters into the text box is returned by the showInputDialog() method as a string, which is usually assigned to a String variable and then processed by the program.

Here is how to call the showInputDialog():

String str = JOptionPane.showInputDialog("Enter Student ID: ");

Example Program

```
// Addition program that uses JOptionPane for input and output.
import javax.swing.JOptionPane; // program uses JOptionPane
public class Addition
{
```

```
public static void main( String args[] )
{
// obtain user input from JOptionPane input dialogs
String firstNumber = JOptionPane.showInputDialog( "Enter first integer" );
String secondNumber = JOptionPane.showInputDialog( "Enter second integer" );
```

```
// convert String inputs to int values for use in a calculation
int number1 = Integer.parseInt( firstNumber );
int number2 = Integer.parseInt( secondNumber );
int sum = number1 + number2; // add numbers
// display result in a JOptionPane message dialog
JOptionPane.showMessageDialog( null, "The sum is " + sum, "Sum of Two Integers", JOptionPane.PLAIN_MESSAGE );
} // end method main
} // end class Addition
```

When the user clicks **OK**, showInputDialog returns to the program the **figure** typed by the user as a String. The program must convert the String to an int.

The GUI Elements

Component	**Description**
JLabel	Displays uneditable text or icons.
JTextField	Enables user to enter data from the keyboard. Can also be used to display editable or uneditable text.
JButton	Triggers an event when clicked with the mouse.
JCheckBox	Specifies an option that can be selected or not selected.
JComboBox	Provides a drop-down list of items from which the user can make a selection by clicking an item or possibly by typing into the box.
JList	Provides a list of items from which the user can make a selection by clicking on any item in the list. Multiple elements can be selected.
JPanel	Provides an area in which components can be placed and organized. Can also

	be used as a drawing area for graphics.

1. Buttons

Three steps must be followed in order to add a push button to the container.

i. Create the push button which is done by declaring an instance of the **JButton** class. With few exceptions, most buttons you create will have a label on them. Passing text to the constructor creates the label. Here's how you'd create a "Start" button:

JButton start = new JButton("Start");

ii. Pass a reference to the instance of JButton to the add() method of the container. Depending on the layout manager you choose, you may also want to specify the location of the button as the second argument to the add() method.

iii. Pass a reference to the content pane of the container to the **setContentPane()** method. You create a content pane by calling the **getContentPane()** method, as shown in the following example.

```
import java.awt.*;
import javax.swing.*;
public class GUIDemo
{
public static void main(String[] args)
{
 Demo dm = new Demo();
 }
} class Demo extends JFrame
{
 public Demo (){
Super("Window Title to be displayed")
setSize(300,150);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setVisible(true);
Container ca = getContentPane();
 ca.setBackground(Color.lightGray);
 FlowLayout flm = new FlowLayout();
```

```
ca.setLayout(flm);
JButton start = new JButton("Start");
ca.add(start);
JButton stop = new JButton("Stop");
 ca.add(stop); setContentPane(ca);
 }
 }
```

2. Labels and Text Fields

Two of the most commonly used GUI elements are the label and the text field. The label element is used to place text in the container, and the text field element enables the person who uses your program to enter text.

You create a label by declaring an instance of the JLabel class and passing its constructor the text that will appear as the label, as shown here, where "Student Information" is the text of the label:

JLabel lab1 = JLabel("Student Information");

You create a text field by declaring an instance of the JTextField class and passing the constructor of this class two arguments. The first argument is the text that will appear in the text field. The second argument is the number of characters that can be entered into the text field.

The following example shows how to add a label and text field to a container. Notice that that add() method is called for each GUI element. Once all the elements have been added to the content pane, the setContentPane()method is called to place the content pane in the container.

```
import java.awt.*;
import javax.swing.*;
public class GUIDemo
{
public static void main(String[] args)
{
GUIDemo win = new GUIDemo ();
 }
}
class GUIDemo extends JFrame
{
public GUIDemo ()
```

```
{
 super ("Window Title");
setSize(400,100);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setVisible(true);
Container ca = getContentPane();
ca.setBackground(Color.lightGray);
FlowLayout flm = new FlowLayout();
ca.setLayout(flm);
JLabel lab1 = new JLabel("Student Information");
ca.add(lab1);
JTextField text = new JTextField("First Name",25);
ca.add(text); setContentPane(ca);
}
}
```

3. Radio Buttons and Check Boxes

Radio buttons and check boxes enable the user to choose a selection rather than having to enter the selection into a text field. Radio buttons are usually displayed in a group. Only one radio button within the group can be selected. All other radio buttons become deselected automatically when the user selects one radio button within the group.

Check boxes are not grouped together, enabling a user to select none, all, or a combination of check boxes.

A radio button is created by declaring an instance of the JRadioButton class. Each radio button is uniquely identified within the group by a label. You create the label by passing text to the JRadioButton constructor, as shown here:

JRadioButton rb1 = new JRadioButton("Pass");

You must also create a radio button group. You do this by declaring an instance of the ButtonGroup class, as shown here:

ButtonGroup passFail = new ButtonGroup();

You add a radio button to the button group by calling the add() method of the instance of the ButtonGroupclass. This is shown in the following statement, where the radio button rb1 is added to the button group passFail:

passFail.add(rb1);

You then add the button group to the content pane by calling the add() method, as shown in previous examples.

Check box by declaring an instance of the JCheckBox class and passing its constructor the text that will be used as the label for the check box. This is shown here:

JCheckBox cb1 = new JCheckBox("Completed");

A reference to the check box is passed to the add() method of the content pane in order for the check box to appear in the content pane. This is basically the same step used to pass a reference to the button group to the add() method of the content pane.

Once all the GUI elements are added to the content pane, the setContentPane() method is called and is passed a reference to the content pane.

4. Combo Boxes

A combo box is a GUI element that enables the user of your program to select an item from a list of items contained in a drop-down menu. You create a combo box by declaring an instance of the JComboBox class, as shown:

JComboBox mycombo = new JComboBox();

Once the instance is declared, you insert items into the drop-down list by calling the addItem() method defined in JComboBox. The **addItem()** method requires one argument: the text of the item you want added to the combo box. The following statement inserts the text “One” into the instance of the JComboBox called combo1.

mycombo.addItem("One");

Two additional steps are necessary to place the combo box in the container of the window:

First, you'll need to place the combo box in the content pane by calling the add() method of the content pane. Second, you'll place the content pane in the container by calling the setContentPane() method.

5. Text Area

The text area GUI element is used to place a block of text in the window. You create a text area by declaring an instance of the class, as shown:

JTextArea ta = new JTextArea("Default text",5, 30);

The constructor of the JTextArea requires two arguments: the number of lines and the number of characters that can appear on each line. The number of lines refer to the height of the text area and the number of characters as its width.

The number of characters you specify is really an approximation made by the Java Virtual Machine because the actual number of characters that fit on a line depends on the font used to display the text.

Another version of the JTextArea() constructor uses three arguments,

- first of which is the text that appears in the text area.
- The other two arguments are the number of lines and the number of characters, which define the height and width of the text area.

You can place text within the text area by calling the **setText()** method and passing it the text you want displayed in the text area. The instance of the JTextArea is called ta:

ta.setText("Default text");

A text area can be used to display text, but you can also use it to have the user of your program enter text or edit text that already appears in the text area. You determine whether the user can edit the text area by calling the setEditable() method and passing it either a Boolean true (to make the text area editable) or a Boolean false (to make the text read-only).

ta.setEditable(true);

The following example shows how to create a text area in a window. This example shows a text area that is five lines high and approximately 30 characters wide

6. Scroll Pane

Sometimes the entire contents of a GUI element won't fit in the space allocated for the element. This is the case when text exceeds the height of a text area. In order to enable the user to see additional contents, you can use a scroll pane. A scroll pane is a GUI element that enables the user to scroll another GUI component both horizontally and vertically by using a scroll bar.
To create a scroll pane, declare an instance of the JScrollPane class, as shown here:

JScrollPane sp=newJScrollPane(ta,JScrollPane.VERTICAL_SCROLLBAR_ALWAYS, JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);

The constructor of the JScrollPane class accepts three arguments.

- **first argument** is a reference to the GUI element that will use the scroll bars. In the preceding statement, ta is a reference to a text area GUI element.
- **second** and **third arguments** are constants of the JScrollPane class that specify the behavior of the vertical and horizontal scroll bars.

The preceding statement causes both the vertical and horizontal scroll bars to always appear, even if all the content of the GUI element appears on the screen.

Revision Exercise

1. Ensure that all the two code examples(button and texbox,label) are compile and executed to produce an output.
2. Write a program to produce a two radio button for student gender for those pursing BSC IT.
3. Write a program code that uses four checkboxes for four units in BSC IT of your choice.
4. Write a program code that uses a scroll pane to contain any information about students.
5. Write a program code that combines all the concepts above together to produce a GUI to help in capturing students details such surname,othernames,course name,age,gender,units.it should also contain ok and cancel buttons.

3. J2ME Mobile development.

Objectives:

At the end of this topic the students should be able to:

- Understand Mobile development.
- Differentiate between configuration and profiles.
- Understand the MIDP.
- Create the MIDlet Application

Introduction to J2ME -Mobile Computing

Sun Microsystems defines J2ME as "a highly optimized Java run-time environment targeting a wide range of consumer products, including pagers, cellular phones, screen-phones, digital set-top boxes and car navigation systems.".**J**ava 2 Platform, Micro Edition (J2ME) is the second revolution in Java's short history.

Java, as a platform, really took off with the advent of *servlets*, Java programs that run on a server (offering a modular and efficient replacement for the vulnerable CGI).
Java further expanded into the server side of things, the better features of Java 2 Platform, Enterprise Edition (J2EE). The second revolution is the explosion of small-device Java.

Understanding J2ME

J2ME is not a specific piece of software or specification instead it is Java for small devices. Small devices range in size from pagers, mobile phones, and personal digital assistants (PDAs) even including set-top boxes that are not fully desktop PCs.

J2ME is divided into ***configurations, profiles,*** and ***optional APIs,*** which provide specific information about APIs and different families of devices.

Configurations

Configuration determines the JVM used, and a profile, which defines the application by adding domain-specific classes.

A configuration is designed for a specific kind of device based on memory constraints and processor power. The *configuration* defines the basic run-time environment as a set of core classes and a specific JVM that run on specific types of devices.

It specifies a Java Virtual Machine (JVM) that can be easily ported to devices supporting the configuration.

It also specifies a strict subset of the Java 2 Platform, Standard Edition (J2SE) APIs that will be used on the platform, as well as additional APIs that may be necessary. Device manufacturers are responsible for porting a specific configuration to their devices.

Currently there are two: the Connected Device Configuration (CDC) and the Connected, Limited Device Configuration (CLDC).

The configurations and profiles of J2ME are generally described in terms of their memory capacity. Usually a minimum amount of ROM and RAM is specified. For small devices, it makes sense to think in terms of volatile and nonvolatile memory. The nonvolatile memory is capable of keeping its contents intact as the device is turned on and off. ROM is one type, but nonvolatile memory could also be flash memory or battery-backed RAM. Volatile memory is essentially workspace and does not maintain its contents when the device is turned off.

Configurations

i. Base configuration for a range of devices
ii. Connected Limited Device Configuration (CLDC)
iii. Connected Device Configuration (CDC)

Profiles

Profiles are more specific than configurations. A profile is based on a configuration and provides additional APIs, such as **user interface**, **persistent storage**, and whatever else is necessary to develop running applications for the device.

The *profile* defines the application; specifically, it adds domain-specific classes to the J2ME configuration to define certain uses for devices. A profile is layered on top of a configuration, adding the APIs and specifications necessary to develop applications for a specific family of devices.

Optional APIs

Defines specific additional functionality that may be included in a particular configuration (or profile).

The whole configuration, profile, and optional APIs—that is implemented on a device is called a ***stack***.

For example, a possible future device stack might be CLDC/MIDP + Mobile Media API.

Currently, there are a handful of configurations and profiles; the most relevant ones for J2ME developers are illustrated in Figure 3-1.

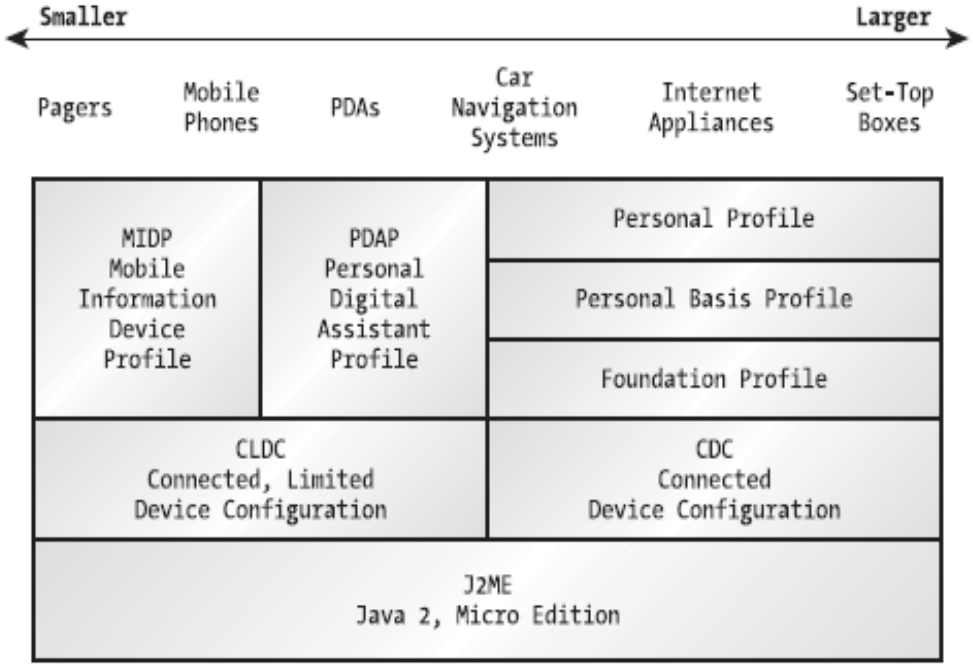


Figure 3-1: Common J2ME profiles and Configurations

Configurations

Name

i. Connected, Limited Device Configuration (CLDC) 1.0
ii. Connected, Limited Device Configuration(CLDC) 1.1
iii. Connected Device Configuration 1.0.1
iv. Connected Device Configuration 1.1

Profiles

Name

i. Mobile Information Device Profile 1.0
ii. Mobile Information Device Profile 2.0
iii. PDA Profile 1.0
iv. Foundation Profile 1.0
v. Personal Basis Profile 1.0
vi. Personal Profile 1.0
vii. Foundation Profile 1.
viii. Personal Basis Profile 1.1

Optional APIs

Name

i. PDA Optional Packages for J2ME
ii. Java APIs for Bluetooth
iii. Mobile Media API
iv. Mobile 3D Graphics
v. Location API for J2ME
vi. Wireless Messaging API 1.0
vii. Wireless Messaging API 2.0
viii. J2ME Web Services APIs
ix. RMI Optional Package

Different Configurations

1. **Connected Device Configuration**

A connected device has, at a minimum, 512KB of read-only memory (ROM), 256KB of random access memory (RAM), and some kind of network connection. The CDC is designed for devices like television set-top boxes, car navigation systems, and high-end PDAs. The CDC specifies that a full JVM must be supported.

2. **Connected, Limited Device Configuration**

CLDC is the configuration is mostly used since it encompasses mobile phones, pagers, PDAs, and other devices of similar size. CLDC is aimed at smaller devices than those targeted by the CDC. The name CLDC appropriately describes these devices,

- Having limited display.
- Limited memory.
- Limited CPU power.
- Limited display size.
- Limited input.
- Limited battery life.
- Limited network connection.

The CLDC is designed for devices with 160KB to 512KB of total memory, including a minimum of 160KB of ROM and 32KB of RAM available for the Java platform.

The "Connected" simply refers to a network connection that tends to be intermittent and probably not very fast. (Most mobile telephones, for example, typically achieve data rates of 9.6Kbps.)

Different Profiles

Currently several different profiles are being developed under the Java Community Process.

The Foundation Profile is a specification for devices that can support a rich networked J2ME environment.

Mobile Information Device Profile

Mobile Information Device has the following characteristics:

- A minimum of 256KB of ROM for the MIDP implementation (this is in addition to the requirements of the CLDC).
- A minimum of 128KB of RAM for the Java runtime heap.
- A minimum of 8KB of nonvolatile writable memory for persistent data.
- A screen of at least 96×54 pixels.

- Some capacity for input, either by keypad, keyboard, or touch screen.
- Two-way network connection, possibly intermittent.

Platform Standardization

Given the profusion of configurations, profiles, and especially optional APIs.

In the next generation of J2ME, a concept called Building Blocks is supposed to replace configurations and profiles.

A *Building Block* is just some subset of a J2SE API. For example, one Building Block might be created from a subset of the J2SE java.io package.

Conceptually, a Building Block represents a smaller chunk of information than a configuration.

Profiles, then,will be built on top of a set of Building Blocks rather than a configuration.

The definition of Building Blocks is a **JSR**

Anatomy of MIDP Applications

The APIs available to a MIDP application come from packages in both CLDC and MIDP, as shown in Figure 3-2. Packages marked with a + are new in CLDC 1.1 and MIDP 2.0, CLDC defines a core of APIs, mostly taken from the J2SE world. These include fundamental language classes in **java.lang**, stream classes from **java.io**, and simple collections from **java.util**.

CLDC also specifies a generalized network API in javax.microedition.io.

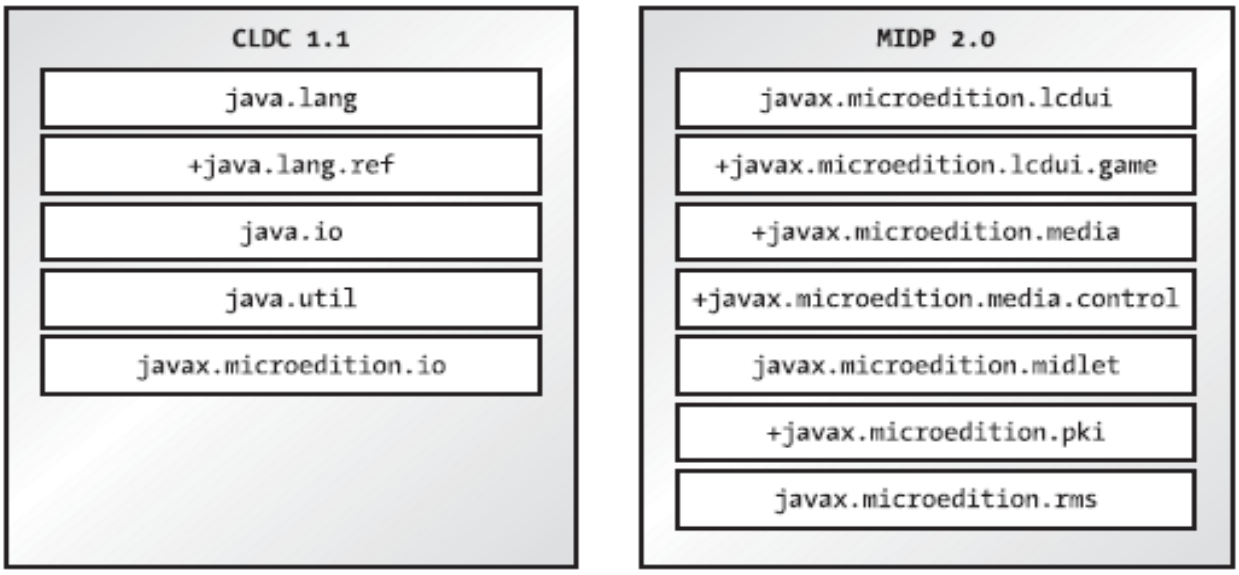


Figure 3-2: MIDP packages

Optionally, device vendors may also supply Java APIs to access device-specific features. MIDP devices, then, will typically be able to run several different flavors of applications.

Figure 3-3 shows a map of the possibilities.

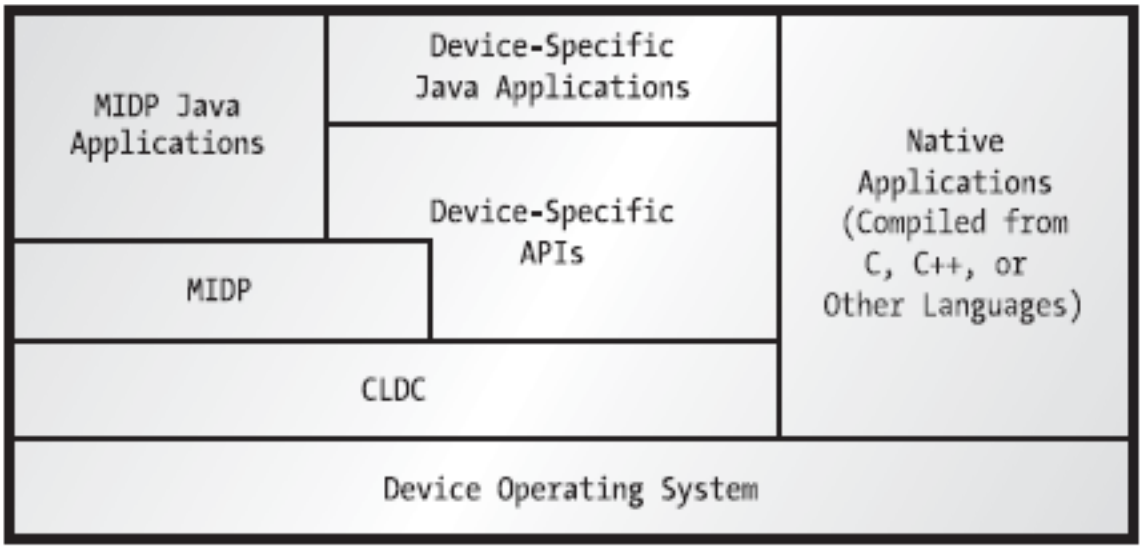


Figure 3-3: MIDP software components.

Each device implements some kind of operating system (OS). Native applications run directly on this layer and represent the world before MIDP—many different kinds of devices, each with its own OS and native applications.

Layered on top of the device OS is the CLDC (including the JVM) and the MIDP APIs. MIDP applications use only the CLDC and MIDP APIs. Device-specific Java applications may also use Java APIs supplied by the device vendor.

Advantages of MIDP

MIDP devices are achieving the kind of processing power, memory availability, and Internet connectivity that makes them an attractive platform for mobile networked applications.

Portability

The advantage of using Java over using other tools for small device application development is portability. You could write device applications with C or C++, but the result would be specific to a single platform. An application written using the MIDP APIs will be directly portable to any MIDP device.

Security

Java has ability to safely run downloaded code like applets. Second, the CLDC does not allow for application-defined classloaders. This means that most dynamic application delivery is dependent on device-specific mechanisms.

MIDP applications do offer one important security promise: they can never escape from the confines of the JVM. This means that, barring bugs, a MIDP application will never be able to write to device memory that doesn't belong to the JVM.

A MIDP application will never mess up another application on the same device or the device OS itself.

In MIDP 2.0, MIDlet suites can be cryptographically signed, and then verified on the device, which gives users some security about executing downloaded code.

Building MIDlets

MIDP applications are normally called MIDlets, a continuation of the naming theme begun by applets and servlets. Writing MIDlets is relatively easy for a moderately experienced Java programmer.

The actual development process, however, is a little more complicated for MIDlets than it is for J2SE applications.

Furthermore,many of the fundamental APIs from **java.lang** and **java.io** are basically the same in the MIDP as they are in J2SE.

Beyond a basic compile-and-run cycle, MIDlets require some additional tweaking and packaging.

The complete build cycle:

i. Edit Source Code
ii. Compile

iii. Preverify
iv. Package
v. Test or Deploy.

MIDlets are developed on regular desktop computers, although the MIDlet itself is designed to run on a small device. To develop MIDlets, you'll need some kind of development kit, either from Sun or another vendor. MIDP is only a specification; vendors are free to develop their own implementations.

Creating Source Code

Writing Java source code is the same as it always was: use text editor to create a source file with a .java extension.

Compiling a MIDlet

Writing MIDlets is an example of cross-compiling, where you compile code on one platform and run it on another. Compiling a MIDlet using J2SE on your desktop computer. The MIDlet itself will run on a mobile phone, pager, or other mobile information device that supports MIDP.

The J2ME Wireless Toolkit takes care of the details as long as you put the source code in the right directory.

- Start the toolkit, called KToolbar.
- Choose New Project from the toolbar to create a new project.
- When the J2ME Wireless Toolkit asks you for the name of the project and the MIDlet class name, use "Gurus" for both.

Preverifying Class Files

Now comes an entirely new step in building your program, *preverifying*. Because the memory on small devices is so scarce, MIDP (actually, CLDC) specifies that bytecode verification be split into two pieces. Somewhere off the device, a preverify step is performed.

The device itself is only required to do a lightweight second verification step before loading classes.

If you are using the J2ME Wireless Toolkit, you don't have to worry about preverifying class files.

Revision Exercise

1. Differentiate between configuration and profiles.
2. Discuss the main characteristics of Connected, Limited Device Configuration
3. Discuss the key characteristics of **Mobile Information Device Profile.**
4. Differentiate between a stack and a building block.
5. Explain three advantages and disadvantages of MIDP.
6. State the uses optional API's in J2ME.

4. Introduction to Midlet Application.

Objectives:

At the end of this topic the students should be able to:

- Understand MIDlet Application requirement.
- Understand how to develop MIDlet application.
- Understand considerations in developing MIDlet Application.
- Understand MIDlet life cycle.
- Create and package MIDlet application.

MIDlet Application environment setup and requirements

MIDP application are called Midlets, Every midlet is an instance of j**avax.microedition.midlet.Midlet**

- It has no argument constructor
- Implements the lifecycle methods

Conceptually Midlets are similar to applets in that:

- ✓ Can be downloaded
- ✓ Executed in the host environment.

One or more MIDlets are packaged together into what is referred to as a ***MIDlet suite,*** composed of:

i. JAR (Java archive) file

ii. JAD (Java Application Descriptor) file

All the user-defined classes and resources required by the suite's MIDlets must be in the JAR file. The JAR file must also include a manifest with a number of MIDP specific entries that describe the MIDlets in the suite.

Developing MIDlets

The reality of MIDP programming today is that the applications you can write are constrained in many ways.

- Memory is a particularly scarce resource.
- Limited screen size.
- Limited processor.
- There are bugs in the implementations of J2ME on some phones
- Operators as the posibility to brand the phones.

Porting applications

Porting and testing applications is very time consuming since there are differences between different phone models such as:

- Screen size
- Memory
- Processor
- Bugs
- Operator problems
- Specification divergence

Tools

To solve the problems with restricted devices and porting you'll need some tools.

- Computer with Windows or Linux.
- JDK (Java development kit).
- WTK (wireless toolkit).
- *IDE (Eclipse, Jbuilder, ...).*
- *A Java phone(java enabled).*
- *JDK, WTK and IDE can be downloaded from suns website or* ***www.eclipse.org***.

Developing J2ME applications

Introduction

When developing application there are some considerations you need to keep in mind when developing applications for smaller devices.

Design considerations for small devices

Developing applications for small devices requires you to keep certain strategies in mind during the design phase. It is best to strategically design an application for a small device before you begin coding. Correcting the code because you failed to consider all of the aspects before developing the application can be a painful process.

- **Keep it simple.** Remove unnecessary features, possibly making those features a separate, secondary application.
- **Smaller is better.** This consideration should be a "no brainer" for all developers. Smaller applications use less memory on the device and require shorter installation times. Consider packaging your Java applications as compressed Java Archive (jar)files.
- **Minimize run-time memory use.** To minimize the amount of memory used at run time, use scalar types in place of object types. Also, do not depend on the garbage collector.

You should manage the memory efficiently yourself by setting object references to null when you are finished with them. Another way to reduce run-time memory is to use lazy instantiation, only allocating objects on an as-needed basis. Other ways of reducing overall and peak memory use on small devices are to release resources quickly, reuse objects, and avoid exceptions.

Design considerations for mobile devices

The same rule of thumb applies for mobile device development that we mentioned for small device development: design and then code.

- **The servers do most of the work.** Move the computationally intensive tasks to the server, and server should run them for you. Let the mobile device handle the interface and minimal amounts of computations. Of course, the mobile device for which you are developing plays a factor in how easy and how often the device can connect to a server.
- **Choose the language carefully.** J2ME still is in its infancy and may not be the best option. Other object-oriented languages, like C++, could be a better choice, depending on your needs.

Performance considerations

Code for performance. Here are some ways to code with the aim to achieve the best performance:

- **Use local variables.** It is quicker to access local variables than to access class members.
- **Avoid string concatenation.** String concatenation decreases performance and can increase the application's peak memory usage.
- **Use threads and avoid synchronization.** Any operation that takes more than 1/10 of a second to run requires a separate thread. Avoiding synchronization can increase performance as well.
- **Separate the model using the model-view-controller (MVC).** MVC separates the logic from the code that controls the presentation.

Compiling considerations

As with any other Java application, you compile the application before packaging and deploying it. With J2ME, however, you use the J2SE compiler and need to invoke it with the appropriate options.

In particular, you need to use the -bootclasspath option to instruct the compiler to use the J2ME classes, not the J2SE classes. Do not place the configuration classes in the compiler's CLASSPATH. This approach will lead to run-time errors, because the compiler automatically searches the J2SE core classes first, regardless of what is in the CLASSPATH.

In other words, the compiler will not catch any references to classes or methods that are missing from a particular J2ME configuration, resulting in run-time errors when you try to run your application.

Packaging and deployment considerations

Because J2ME is designed for small devices with limited memory, much of the usual Java preverification has been removed from the virtual machine to allow for a smaller footprint. As a result, it is necessary to preverify your J2ME application *before* deployment. An additional check is made at run time to make sure that the class has not changed since preverification.

Exactly how to perform the preverification, or checking the classes for correctness, depends on the toolkit. CLDC comes with a command-line utility called preverify, which does the actual verification and inserts extra information into the class files. MIDP uses the wireless toolkit, which comes with a GUI tool, though this too can be run from the command line.

Deployment depends on the platform to which you are deploying. The application must be packaged and deployed in a format suitable for the type of J2ME device, as defined by the profile.

The MIDlet Life Cycle

MIDP applications are represented by instances of the **javax.microedition.midlet.MIDlet** class. MIDlets have a specific life cycle, which is reflected in the methods and behavior of the MIDlet class.

A piece of device-specific software, the ***application manager***, controls the installation, execution, and life cycle of MIDlets. MIDlets have no access to the application manager. A MIDlet is installed by moving its class files to a device. The class files will be packaged in a Java Archive (JAR), while an accompanying descriptor file (with a .jad extension) describes the contents of the JAR.

MIDlet goes through the following states:

- When the MIDlet is about to be run, an instance is created. The MIDlet's constructor is run, and the MIDlet is in the *Paused* state.
- Next, the MIDlet enters the *Active* state after the application manager calls startApp(). While the MIDlet is Active, the application manager can suspend its execution by calling pauseApp(). This puts the MIDlet back in the Paused state. A MIDlet can place itself in the Paused state by calling notifyPaused().
- While the MIDlet is in the Paused state, the application manager can call startApp() to put it back into the Active state.
- The application manager can terminate the execution of the MIDlet by calling destroyApp(), at which point the MIDlet is *destroyed* and patiently awaits garbage collection. A MIDlet can destroy itself by calling notifyDestroyed()

Summary of the midlet lifecylce

A MIDlet lifecycle have following steps...

- **startApp()**
- **pauseApp()**
- **destroyApp()**

Constructor – gets called once, when MIDlet is first created.

startApp() – get called the first time the MIDlet is invoked and everytime the MIDlet resumes from a suspended state.

pauseApp() – is called if the user changes focus to another phone application or phone event (e.g. phone call) requires the MIDlet to be suspended.

destroyApp(boolean arg0) – is called if the user chooses to exit the application (e.g. presses red home button) or if the phone needs more resources for another process. You can throw an exception to say you'd really not like to quit, but the platform can override.

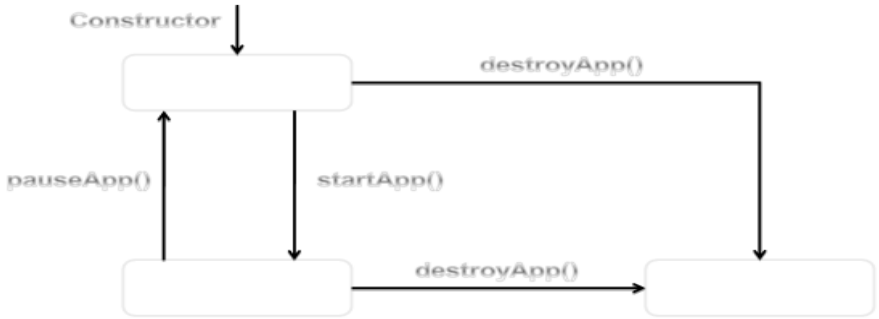


Fig View one :

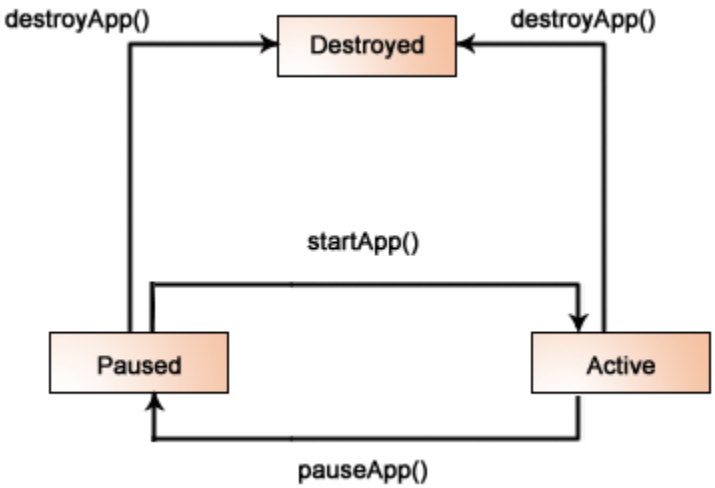


Fig View two

Fig: MIDlet Lifecycle

By default MIDlet is in the paused states. When the application is executed, by default **startApp()** method will be called and when you close the application the **destroyApp()** method will be called. But when your constructor is not null type then it will be executed firstly.

Creating MIDlet Application.

Use step 1 to step 4 to aid in creating the application in Netbeans 6.5.1 or above

Caution: Note that some higher versions of netbeans 7.0 might not support the J2ME application.

Step 1: File->New Project

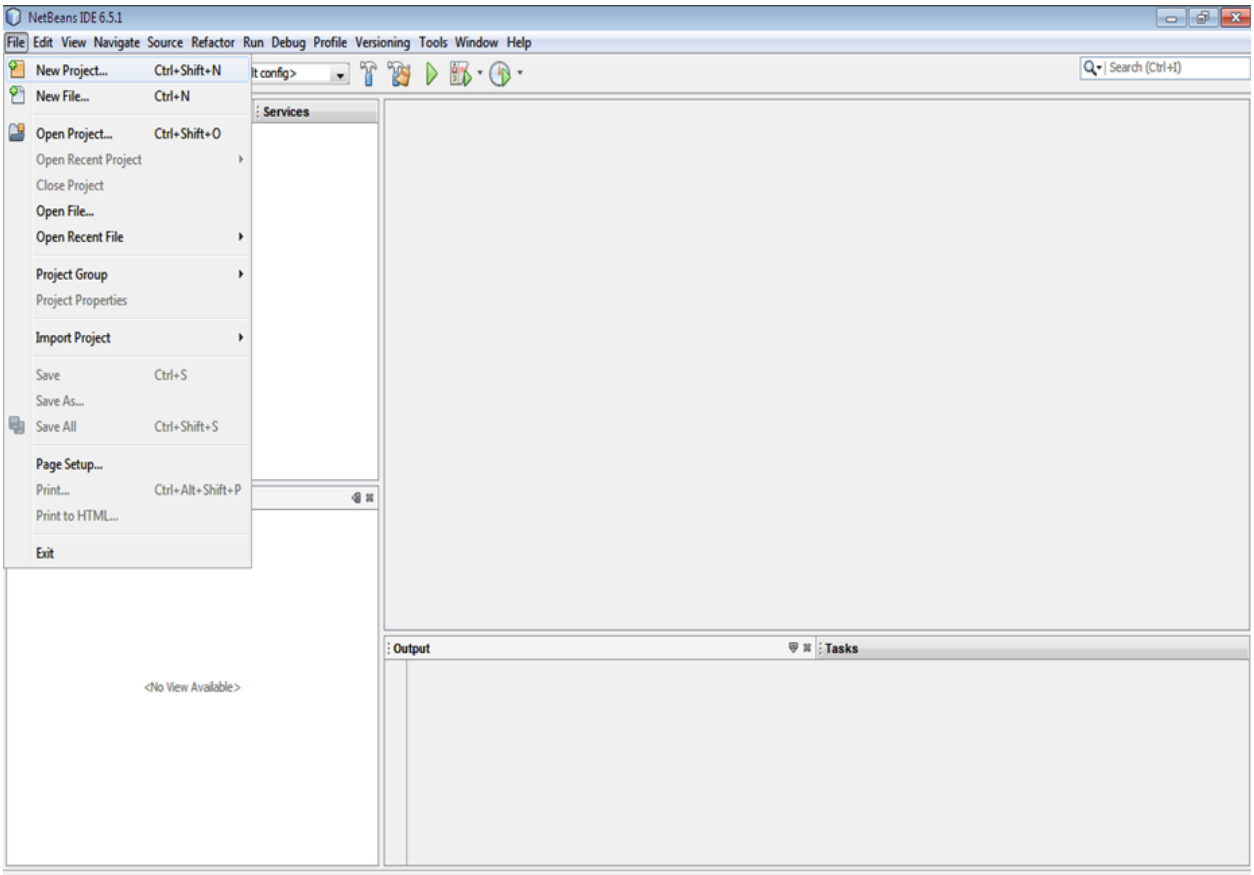


Figure 4-1:Step 1.

Step2: A dialog box will appear with two window(Categories and Projects)click on JavaME on category window, then click mobile application.

On the project window, click next button

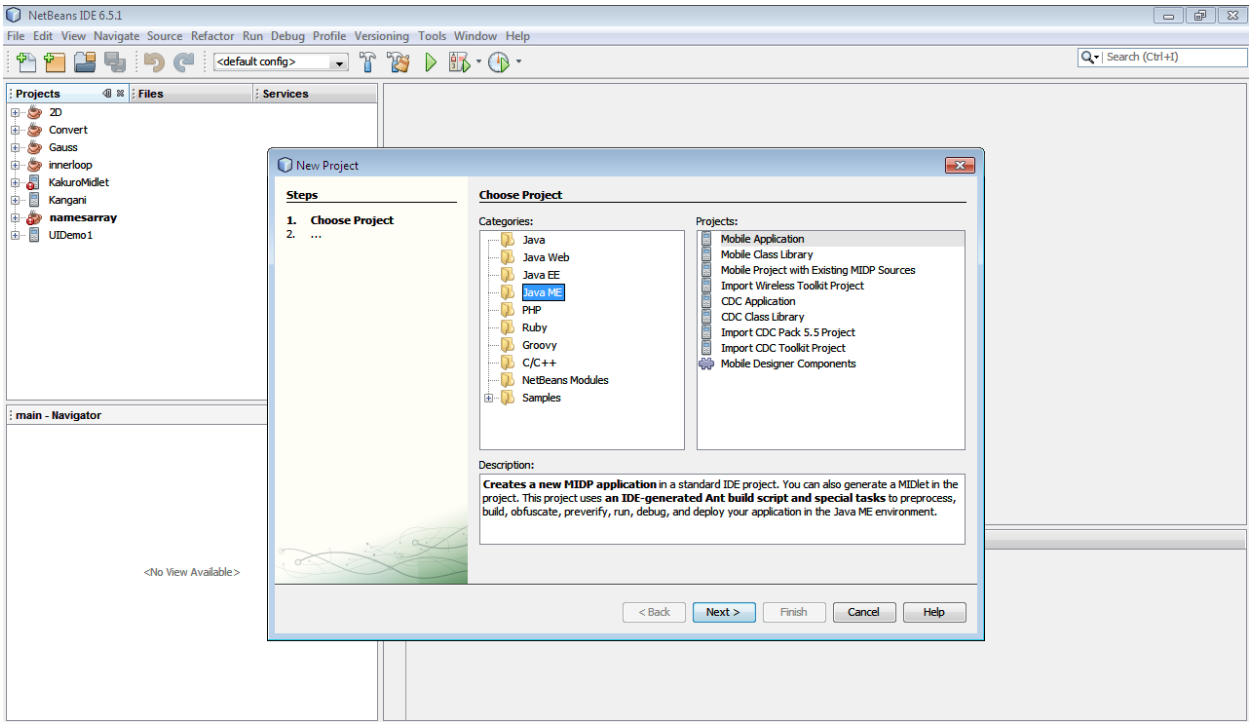


Figure 4-2: Step 2.

Step 3: On the pop window enter the name of the project,Uncheck create hello midlet or leave it, click finish.

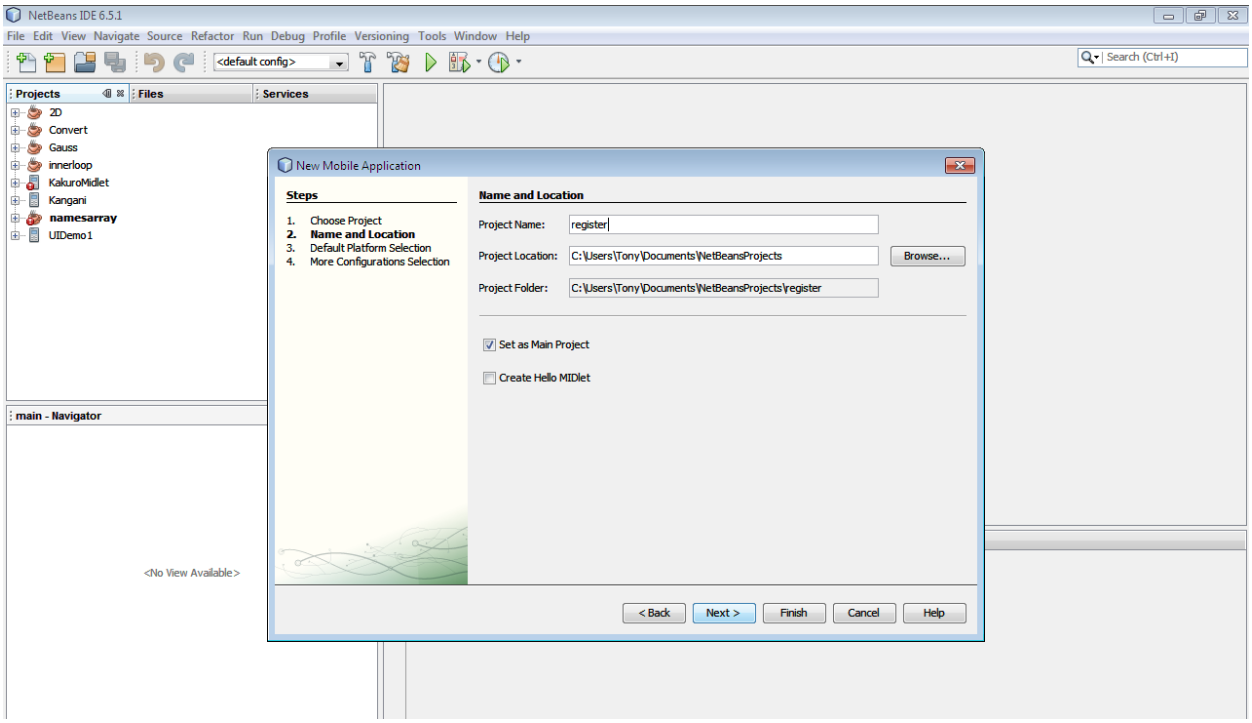


Figure 4-3: Step 3

Step 4: After clicking the finish button a mobile phone icon will Be displayed on the left window, highlight and right click, on source packages, then choose new|MIDlet.

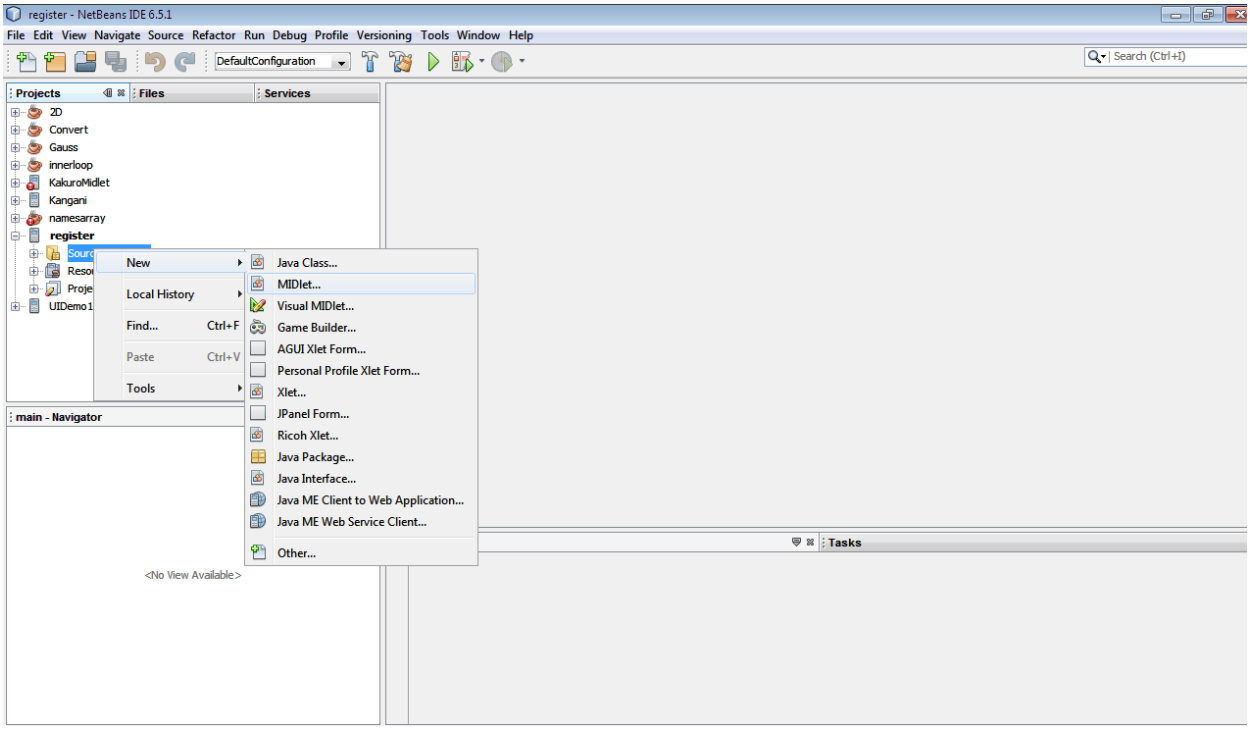


Figure 4-4: Step 4.

Step 5: Enter class name on the pop up window, then click Finish button.

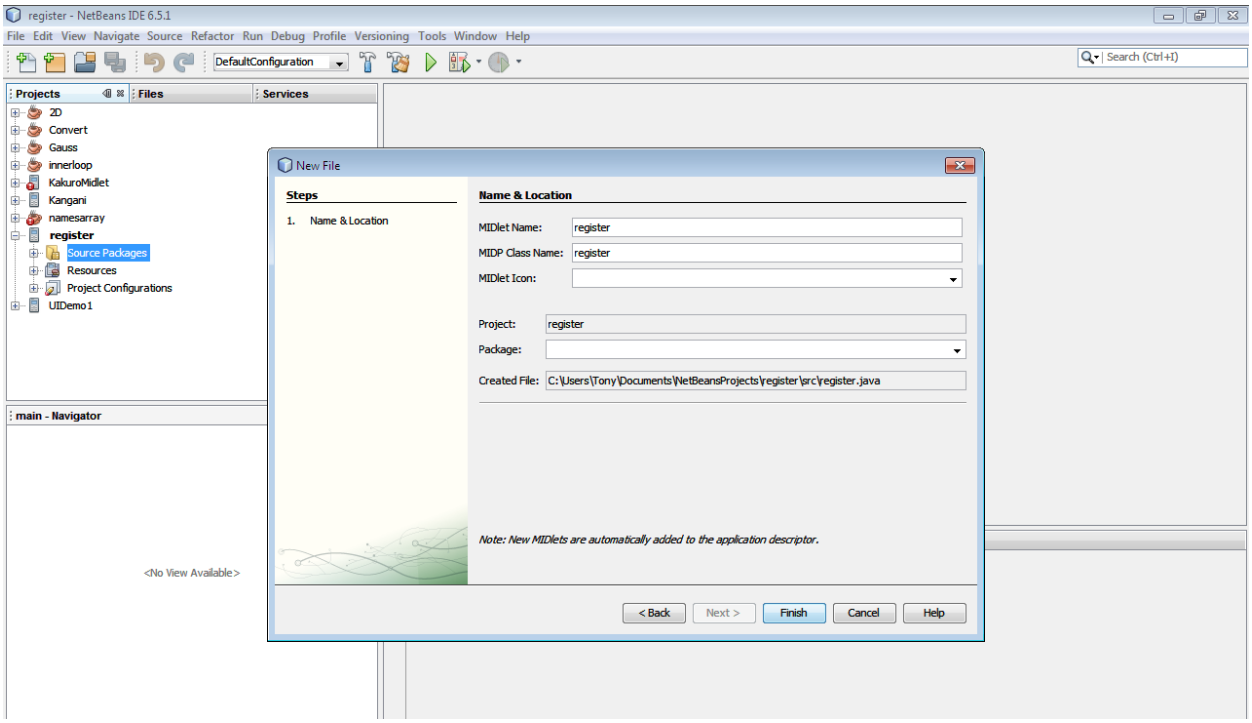


Figure 4-5: Step 5.

On the right window you will get predefined codes with the

Following;

a. Imported Package.

b. Class name

c. Methods(startApp,PauseApp,destroyApp)

The source code of life cycle execution is as follows:

```
import javax.microedition.lcdui.*;
import javax.microedition.midlet.MIDlet;
public class MidletLifecycle extends MIDlet{
private Form form;
private Display display;
public MidletLifecycle(){
```

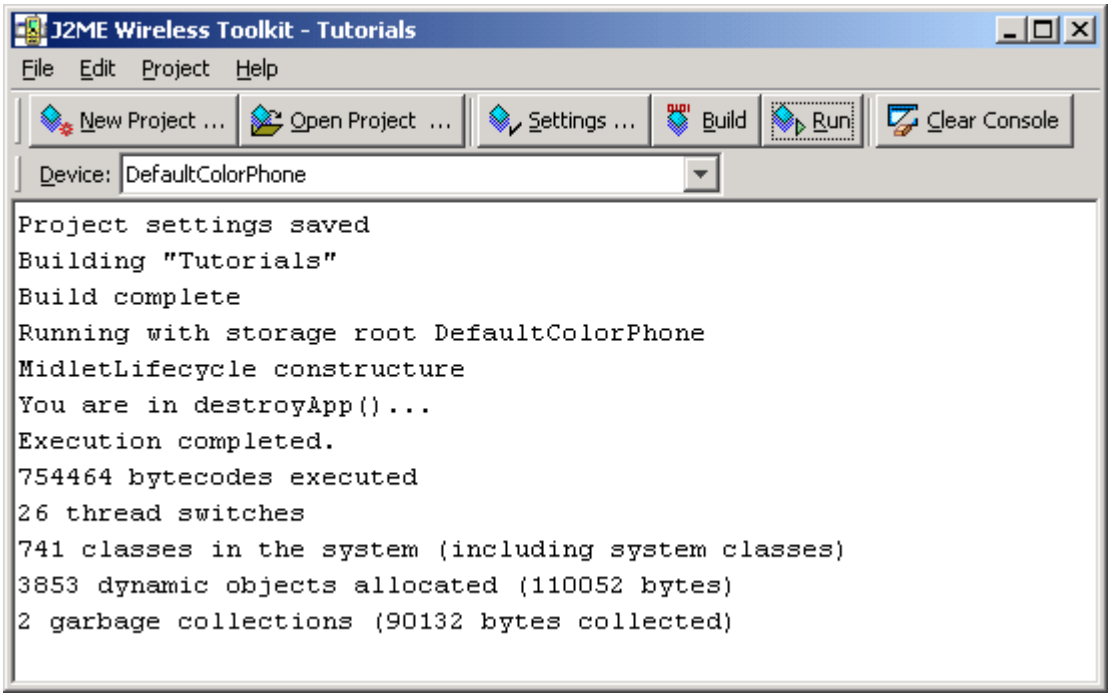
```
  System.out.println("MidletLifecycle constructure");
  }
  public void startApp()

{
 form = new Form("Midlet Lifecycle");
 display = Display.getDisplay(this);
 String msg = "This is the Lifecycle of Midlet!";
 form.append(msg);
 display.setCurrent(form);
  }
  public void pauseApp()

{
  System.out.println("You are in pauseApp()...");
  }
  public void destroyApp(boolean destroy)

{
  System.out.println("You are in destroyApp()...");
  notifyDestroyed();
  }
}
```

Output:



Source code of 'jad' and 'properties' file

Java Application Descriptor (JAD) filename extension is .jad and media type is text/vnd.sun.j2me.app-descriptor, which developed by the Sun Microsystems, Inc. The JAD file is used for java or games application. The java application enabled mobile phone connected programmatically with online Web Services. Through this facility we can send SMS via GSM mobile Internet.

Packaging MIDlets

MIDlets are deployed in *MIDlet suites*. A MIDlet suite is a collection of MIDlets with some extra information. There are two files involved. One is an *application descriptor*, which is a simple text file. The other is a JAR file that contains the class files and resource files that make up your MIDlet suite. Like any JAR file, a MIDlet suite's JAR file has a manifest file.

A simpe hello there program

The Application is as follows:

HelloWorld.java// name of the program

The source code

```
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;
public class HelloWorld extends MIDlet{
private Form form;
private Display display;
public HelloWorld(){
super();
  }
  public void startApp(){

 form = new Form("Hi there");

String msg = "Say something !!!!!!!";

form.append(msg);
display = Display.getDisplay(this);
display.setCurrent(form);
  }
  public void pauseApp(){}
  public void destroyApp(boolean unconditional){
  notifyDestroyed();
  }
}
```

This is the simple hello world application. In this example we are creating a form name "Hi there" and creating a string message "Hello World!!!!!!!" as below:

Form form = new Form("Hi there");

String msg = "Say something";

In this application, To display the message we are using append method with the form as below:

form.append(msg);

Compiling a MIDlet

Writing MIDlets is an example of cross-compiling, where you compile code on one platform and run it on another. In this case, you'll be compiling a MIDlet using J2SE on your desktop computer. The MIDlet itself will run on a mobile phone, pager, or other mobile information device that supports MIDP.

The J2ME Wireless Toolkit takes care of the details as long as you put the source code in the right directory.

Running MIDlets

Sun's MIDP reference implementation includes an emulator named midp. It emulates an imaginary MID, a mobile telephone with some standard keys and a 182×210-pixel screen. The J2ME Wireless Toolkit includes a similar emulator, as well as several others.

Once the class file is preverified, use the midp emulator to run it. The emulator is an application that runs under J2SE and acts just like a MIDP device. It shows itself on your screen as a representative device, a generic mobile phone.

SUN's J2ME Wireless Toolkit emulator exhibits several qualities that you are likely to find in real devices:

The device has a small screen size and limited input capabilities.

Two ***soft buttons*** are available. A soft button does not have a fixed function. Generally, the function of the button at any given time is shown on the screen near the button. In MIDlets, the soft buttons are used for commands.

Navigation buttons are provided to allow the user to browse through lists or other sets of choices.

A ***select button*** allows the user to make a choice after moving to it with the navigation buttons.

The MIDlet life cycle manage the flow of application. It is in the **javax.microedition.midlet** package, so import this package in your application. The **javax.microedition.icdui** package is used for following classes:

- Alert
- AlertType
- Canvas
- ChoiceGroup
- Command
- DateField
- Display
- Displayable
- Font
- Form
- Gauge
- Graphics
- Image
- ImageItem
- Item
- List
- Screen
- StringItem
- TextBox
- TextField
- Ticker

Revision Exercise.

1. Define the MIDlet application and explain how it is similar to applets.
2. Name two main files found in the MIDlet suite.
3. Differentiate between design consideration for small devices and mobile device.
4. Explain the functions of application manager in midlet application.
5. Discuss performance considerations and compiling considerations.
6. Discuss the MIDlet application life-cycle.
7. Write a simple MIDlet application to display your name and Registration number on the screen.
8. Explain how to package MIDlet application.
9. Explain how buttons are grouped in MIDlet Application.

5. Low-level and High-level IU in MIDlet Application

Objectives:

At the end of this topic the students should be able to:

- Understand the difference between Low-level and High-level IU.
- Create MIDlet application for Low-level and High-level IU.
- Create MIDlet GUI application.

Introduction

User Interface elements are used to interact with the user. Commands are used to initiate actions e.g. navigating to the next page, sending/receiving information from server etc. A Display object manages the device display and controls what is shown on the device.

MIDP 2.0 provides UI classes in the following packages;
javax.microedition.lcdui and
javax.microedition.lcdui.game
javax.microedition.lcdui package can be divided into two;

1. High level UI s;

- Form
  - Items of a form
- Alert
- List
- TextBox

2. Low level UI s.

- Canvas
- Graphics

J2ME display and displayable

A **Display object** is the manager of the device display, and controls what is to be shown on the screen.

A **Displayable object** is a component that is visible on the screen and can also contain items and commands e.g. a form. Each MIDlet consists of one Display object and any number of Displayable objects.

The Displayable Hierarchy

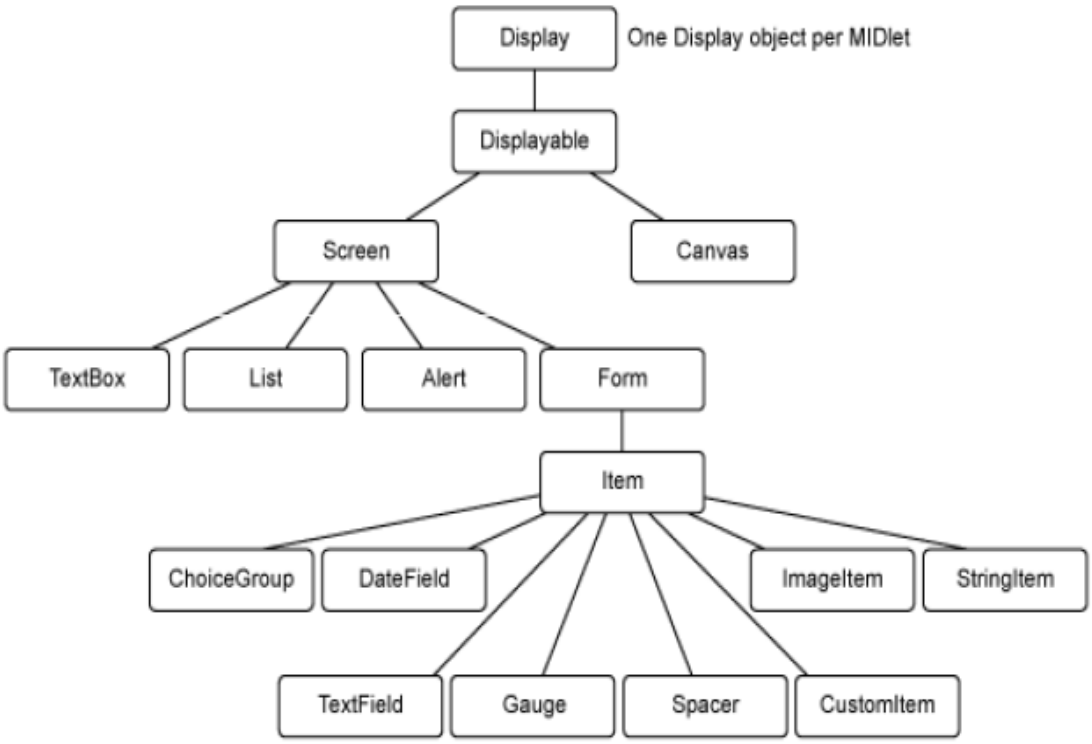


Figure 5-1: Displayable object Hierarchy.

The Screen sub-classes are abstract, meaning it is up the MIDP implementation to decided on their appearance. All these classes are defined in javax.microedition.lcdui.

Graphical user interface includes

Included in javax.microedition.lcdui.*

1. "Form" which contains Basic items :
    - **ChoiceGroup** -A ChoiceGroup is a group of selectable elements intended to be placed within a Form.
    - **ImageItem** –Hold an image for display.

- **StringItem** – display text.
- And some others like CustomItem, Gauge, DateField, Textfield, Spacer, ImageItem…

Form is used for simple screen output and text input.

Figure 5-2: A simple calendar application develop by Form

2. Canvas which contains Graphics, you are able to

- Draw images and strings
- Draw rectangles, lines and arcs
- Set the color used
- Canvas also allows you to get “key” input from user.

Note, Form are used for some simple applications only have text input while Canvas are used for more interactive applications like games

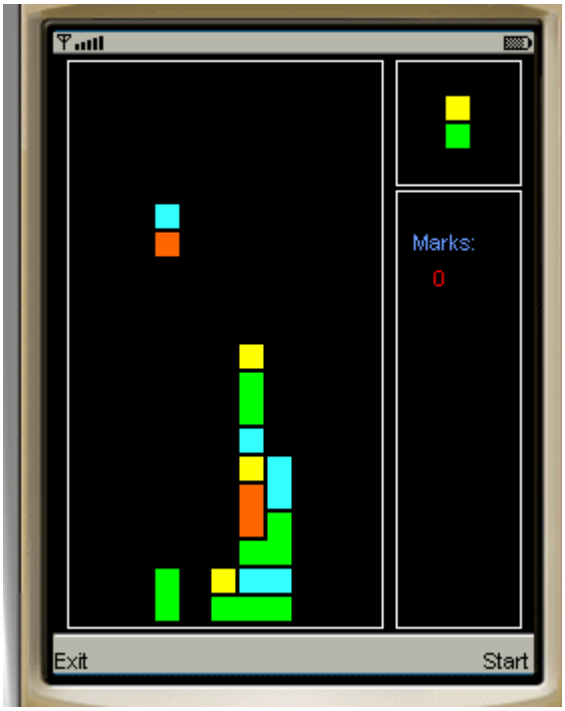


Figure 5-3: A puzzle game developed by Canvas.

The Low-level IU.

Textfield

The Allows the user to enter editable text.

1. Constraints to Limit input
    - ANY allows any type of input.
    - NUMERIC restricts the input to numbers.
    - DECIMAL allows numbers with fractional parts.
    - PHONENUMBER requires a telephone number.
    - EMAILADDR input must be an e-mail address.
    - URL input must be a url.
2. Constraints to determine display (Flags)
    - UNEDITABLE: Text is not editable.

- PASSWORD: Text is obscured from user.
- SENSITIVE: Text cannot be kept in dictionary(T9) or auto-completed.
- NON_PREDICTIVE: Words not available in dictionary to allow prediction.
- INITIAL_CAPS_WORD: Initial word should be capitalized.
- INITIAL_CAPS_SENTENCE: Initial word of each sentence should be capitalized.
- Both constraints can be combined using the OR Operator.

E.g.

```
TextField txtPassword = new
TextField(“Password”,””,10,TextField.ANY|TextField.PASSWORD)
```

Create Textfield and Button Example

```
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;
public class OursTextF1 extends MIDlet implements CommandListener
{
private Form form;
private Display display;
private TextField Studname, Adno;
private Command ok;
public OursTextF1()
{
Studname = new TextField(“Student Name:", "", 30, TextField.ANY);
Adno = new TextField(“Admission number:", "", 30, TextField.ANY);
ok = new Command("OK", Command.OK, 2);
        }

public void startApp(){
display = Display.getDisplay(this);
Form form = new Form("Text Field");
```

```
form.append(studname);
form.append(Adno);
form.addCommand(ok);
form.setCommandListener(this);
display.setCurrent(form);
}
public void pauseApp(){
}
public void destroyApp(boolean destroy)
{ notifyDestroyed(); }
public void OnyeshaData(){
display = Display.getDisplay(this);
String nam = studentname.getString();
String adn = adno.getString();
Form form = new Form(“Enter some Value"); form.append(nam);
form.append(adn);
display.setCurrent(form);
}
public void commandAction(Command c, Displayable d) {
String label = c.getLabel();
if(label.equals("OK"))
{
OnyeshaData();
}
}
}
```

J2ME-Commands

A command is something the user can invoke.

Figure 5-1: Commands.

We don't really care how it is shown on the screen

Example:

Command c = new Command("OK", Command.OK, 0);

You can add commands to a Displayable using:

public void ddCommand(Command)

Responding to Command Events

When a Command is invoked by the user, a method is called to service the command the exact method is:

public void commandAction(Command c, Displayable d)

c is the Command invoked and d is the Displayable the Command was added to.

We need to tell the Displayable the object in which to call **commandAction()**

Two Steps:

The class of the object must implement the interface **CommandListener**

CommandListener defines **commandAction()**

You tell the Displayable which object by calling **setCommandListener(CommandListener)** on the Displayable

Example.

```
import javax.microedition.lcdui.*;
import javax.microedition.midlet.MIDlet;
public class HelloWorld extends MIDlet implements CommandListener {
private static Command CMD_EXIT = new Command("Exit", Command.EXIT, 0);
private static Command CMD_NEXT = new   Command("Next", Command.OK, 0);
private TextBox textBox1;
private TextBox textBox2;
public void startApp()
{
textBox1 = new TextBox("TextBox1", "The first Displayable", 30, TextField.ANY);
textBox1.addCommand(CMD_NEXT);
textBox1.setCommandListener(this);
textBox2 = new TextBox("TextBox2", "The second Displayable", 30, TextField.ANY);
textBox2.addCommand(CMD_EXIT);
textBox2.setCommandListener(this);
Display.getDisplay(this).setCurrent(textBox1);
 }
public void commandAction(Command c, Displayable d)
{
        if (d == textBox1 && c == CMD_NEXT)
            Display.getDisplay(this).setCurrent(textBox2);
        else if (d == textBox2 && c == CMD_EXIT)
            notifyDestroyed();
}
public void pauseApp()
{
}
public void destroyApp(boolean u)
{
}
```

Form class

A form- is a collections of instances of the Item interface. The TextBox class- is a standalone UI element, while the TextField *is* an Item instance.

Essentially, a textbox can be shown on a device screen without the need for a form, but a text field requires a form.

An item is added to a form using the **append(Item item)** method. The first added item is at index 0, the second at index 1, and so on.

insert(int index, Item newItem) method inserts an item at a particular position. **set(int index, Item newItem)** replaces an item at a particular position specified by the index.

There are eight Item types that can be added to a form.

StringItem: A label that cannot be modified by the user. This item may contain a title and text, both of which may be null to allow it to act as a placeholder. The Form class provides a shortcut for adding a StringItem, without a title: append(String text)

DateField: Allows the user to enter date/time in one of three formats: DATE, TIME, or DATE_TIME.

1. **TextField:** Same as a TextBox.

ChoiceGroup: Same as a List.

Spacer: Used for positioning UI elements by putting some space between them. This element is an invisible UI element and can be set to a particular size.

Gauge: A gauge is used to simulate a progress bar.

ImageItem: An item that holds an image. Like the StringItem, the Form class provides a shortcut method for adding an image: **append(Image image).**

CustomItem: CustomItem is an abstract class that allows the creation of subclasses that have their own appearances, their own interactivity, and their own notification mechanisms. If you require a UI element that is different from the supplied elements, you can subclass CustomItem to create it for addition to a form.

Alerts

An alert is an informative message shown to the user. In the MIDP universe, there are two flavors of alert:

A timed alert is shown for a certain amount of time, typically just a few seconds.
A modal alert stays up until the user dismisses it.
The title must be set while creating the alert and it cannot be changed afterwards: **Alert(“Alert!");**

To set the message the alert displays use, setString(“print message"). **setTimeout(int time)** is used to set the time (in milliseconds) for which the alert is displayed on the screen. If you pass **Alert.FOREVER** as the value of time, you will show the alert forever and make the alert a modal dialog.
There are five types of alerts defined by the class AlertType: **ALARM**, **CONFIRMATION**, **ERROR**, **INFO**, and **WARNING**. These have different looks and feels and can have a sound played along with the alert. You can associate an image with the alert using the method setImage(Image img);

Alert Example

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.Alert;
import javax.microedition.lcdui.Display;
public class  AlertExample extends MIDlet {
Alert alrt;
public AlertExample() {
alrt = new Alert(“Alaalaahhaha!");
alrt.setString(“Mazeeh ukosawa kweli?”);
}
public void startApp() {
Display.getDisplay(this).setCurrent(alrt);
 }
public void pauseApp() {}
public void destroyApp(boolean unconditional) { } }
```

Example of command and Alert

```
import javax.microedition.lcdui.*;
 public class CndExample extends MIDlet implements CommandListener{
 private Form form;
 private Display display;
 private Command okay, goback, cancel, close, help, item, screen, halt;
 public CndExample(){
 form = new Form("Command Form");
 screen = new Command("SCREEN", Command.SCREEN, 1);
 goback = new Command("BACK", Command.BACK, 2);
 cancel = new Command("CANCEL", Command.CANCEL, 3);
 okay = new Command("OK", Command.OK, 4);
 help = new Command("HELP", Command.HELP, 5);
 halt= new Command(“HALT", Command.STOP, 6);
 exit = new Command("EXIT", Command.EXIT, 7);
 item = new Command("ITEM", Command.ITEM, 8);
 }
 public void startApp(){
 display = Display.getDisplay(this); form.addCommand(screen);
 form.addCommand(goback);
  form.addCommand(cancel);
 form.addCommand(okay);
  form.addCommand(help);

 form.addCommand(halt);

 form.addCommand(exit);
 form.addCommand(item); form.setCommandListener(this); display.setCurrent(form);
 public void pauseApp(){}
 public void destroyApp(boolean destroy)
 {
 notifyDestroyed();
 }
```

List

They allow the user to choose an item from a choice of several.

Types of lists;

- MULTIPLE: Multiple options may be selected simultaneously.
- EXCLUSIVE: Only a single option may be selected at a go. The user selects then confirms the choice.
- IMPLICIT: A single selection list which combines selection and confirmation.

List supports the selection of a single element or of multiple elements.

When the user makes a selection in an IMPLICIT list, the commandAction() method of the list's CommandListener is invoked. A special value is passed to commandAction() as the Command parameter:

public static final Command SELECT_COMMAND

For example, you can test the source of command events like this:

```
public void commandAction(Command c, Displayable s) {
if (c == nextCommand)
// ...
else if (c == List.SELECT_COMMAND)
// ...
}
```

Creating Lists in J2me

To create a List, specify a title and a list type. If you have the element names and images available ahead of time, you can pass them in the constructor:

```
public List(String title, int type)
public List(String title, int type,
String[] stringElements, Image[ ] imageElements)
```

The stringElements parameter cannot be null but can be empty; however, image Elements may contain null array elements.

If the image associated with a list element is null, the element is displayed using just the string. If both the string and the image are defined, the element will display using the image and the string.

List array-elements

You will note that some Lists may have more elements than can be displayed on the screen,the actual number of elements that will fit varies from device to device. :

List implementations automatically handle scrolling up and down to show the full contents of the List.

The List implementation handles scrolling automatically so the developer need not worry since they have no control.

List example

```
import javax.microedition.lcdui.List;
import javax.microedition.lcdui.Choice;
import javax.microedition.lcdui.Display;
import javax.microedition.midlet.MIDlet;
public class OurTowns extends MIDlet {
 List KenCity;
 public OurTowns() {
KenCity = new List("Select  a town",Choice.MULTIPLE);
KenCity.append(“Nairobi", null);
KenCity.append(“Eldoret", null);
KenCity.insert(1, “Kisumu", null);
 // inserts between Nairobi and Eldoret
public void startApp() {
        Display display = Display.getDisplay(this);
        display.setCurrent(KenCity);
                }
        public void pauseApp() {
        }
        public void destroyApp(boolean unconditional) {
        }
}
```

Date Field

Inorder to use this element we will use two textFields i.e:datein and dateout and then initialize to 0.

When application will run, first of all the startApp() method will be called and it will display the form with datein and dateout field. For the specific date,java.util.Date class and java.util.TimeZone class are used to show the format of date like Mon,14 Oct,1999.

When <date> is selected, a new window will open with a calendar as given below.

Date Field Example

```
import javax.microedition.lcdui.*;
import javax.microedition.midlet.MIDlet;
import java.util.Date; import java.util.TimeZone; public class DateZone extends MIDlet
 {
private Form form; private Display display;
private DateField datein, dateout;
private static final int DATE = 0;
public DateZone(){ datein = new DateField("Date In:", DateField.DATE,
TimeZone.getTimeZone("GMT"));
dateout = new DateField ("Date Out:", DateField.DATE,
TimeZone.getTimeZone("GMT")); } public void startApp(){
display = Display.getDisplay(this);
Form form = new Form(“Using Date Field Element");
form.append(datein);
form.append(dateout);
display.setCurrent(form);
} public void pauseApp(){ }
public void destroyApp(boolean destroy)
{
notifyDestroyed();
 }
 }
```

J2ME-Gauges

Gauges are UI instances can be represented in non-interactive and interactive mode.

A gauge in a non-interactive mode typically can be used to represent the progress of a certain task; for example, when the device may be trying to make a network connection or reading a datastore, or when the user is filling out a form.

Non-interactive gauges take four values:

The **label -> String**

Whether it is interactive **mode -> true | false**

Maximum value

Initial value

If you don't know how long a particular activity is going to take, you can use the value of INDEFINITE for the maximum value. Note: You cannot create an interactive gauge with an INDEFINITE maximum value.

This type of gauge can be in one of four states, and this is reflected by the initial value.

These states are

- CONTINUOUS_IDLE.
- INCREMENTAL_IDLE.
- CONTINUOUS_RUNNING.
- INCREMENTAL_UPDATING.

CheckBox ChoiceGroup

Offers the user a list of choices to select from.

Types

- Multiple: Multiple options may be selected simultaneously.
- Exclusive: Only a single option may be selected at a go. The user selects then confirms the choice.
- Popup: Displays like a drop down menu or combobox.

We are creating a Form named programming languages with 5 choices, ChoiceGroup ("Java", "C++", "Mobile Application", "PHP","Perl"). If user select a check box and click on "Choose" button then the selected item will be display on the form.

We use StringItem to display the selected item on the form.

We create a variable of array type ("language [ ]") and stored the size of selected item in it.

Next we create a boolean type array ("**sel[**]") to store the size of selected item, we have given a condition if **sel[]** returns true then this will display the **StringItem()** with message and String value of selected item. The StringItem has only 2 methods that is:

getText()

setText(String text)

Constructor StringItem Syntax is:

StringItem(String label, String text)

Revision Exercise

1. Write a J2ME application to create gauge that would be used to estimate a given process progress.
2. Write a MIdlet application to create a list of all the subjects pursued in your year of study.
3. Differentiate between list element and checkbox Choice group.
4. Name two main methods of list object.
5. State two methods of a StringItem.
6. Write a MIDlet application to create a datefield to display the date of today.
7. Name two constraint supported by a textfield in MIDlet application.
8. Differentiate between Low-level and High-level elements.

6. Persistent storage on small hand-held devices-RMS

Objectives:

At the end of this topic the students should be able to:

- Understand Persistent storage.
- Understand how Record Management System.
- Create MIDlet GUI application that uses RMS.

Introduction

The record management system provides a mechanism for midlets to persistently store data and retrieve it later even if the device is switched off. It stores data in binary format in mobile device.

This persistent storage mechanism can be viewed as a simple record-oriented database model.

The RMS uses a storage mechanism called record store. A record store is a very simple database, where each row consists of two columns: one containing a unique row identifier, the other a series of bytes representing the data in the record.

Record ID	Data
1	Array of bytes
2	Array of bytes
...	
...	

The **record store** contains pieces of data called records.

A record store can be limited to a single MIDlet suite or can be shared between MIDlet suites.

Record stores are identified by name. All characters are case sensitive. No two record stores within a MIDlet suite (that is, a collection of one or more MIDlets packaged together) may contain the same name.

Record store implementations ensure that all individual record store operations are atomic, synchronous, and serialized, so no corruption of data will occur with multiple accesses. The record store is timestamped to denote the last time it was modified. The record store also maintains a *version*, which is an integer that is incremented for each operation that modifies the contents of the record store. Versions and timestamps are useful for synchronization purposes.

CLASSES REQUIRED

1. **javax.microedition.rms.***

Contains methods for creating, closing a record store, adding record to the record store among others.

2. **java.io.ByteArrayOutputStream**

This class implements an output stream in which the data is written into a byte array. The buffer automatically grows as data is written to it.

3. **java.io.DataOutputStream**

A data input stream lets an application write primitive Java data types to an output stream in a portable way. DataOutputStream writes strings and primitive data types to an output stream

Strings are written using one of two methods. You can write strings of up to 65,535 characters encoded in UTF-8 format with writeUTF(), or call writeChars() to write a sequence of two-byte characters.

Strings can be read using the readUTF() and readChars() methods.

4. **java.io.ByteArrayInputStream**

Contains an internal buffer that contains bytes that may be read from the stream.

5. **java.io.DataInputStream**

A data input stream lets an application read primitive Java data types from an underlying input stream in a machine-independent way. An application uses a data output stream to write data that can later be read by a data input stream. DataInputStream transforms a raw input stream into primitive data types and strings:

METHODS IN THE RECORD STORE CLASS

- openRecordStore()
- closeRecordStore()
- deleteRecordStore()
- enumerateRecords()- Used to obtain enumeration object, which represents entire set of records in Record Store
- getName()- Used to obtain name of Record Store
- getRecord()- Retrieve a record from Record set
- getNumRecord()- Obtain a number of records in Record Store
- addRecord()- Add record to Record Store
- deleteRecord()- Delete record from Record Store

MANAGING RECORD STORES

Opening or creating a record store

Public static RecordStore openRecordStore(String recordStoreName, boolean createIfNecessary) The parameters allow you to give your database a name, and create a database if it doesn't yet exist

If the record store does not exist, and the createIfNecessary parameter is false, then a RecordStoreNotFoundException will be thrown.

RecordStore myRcrds = RecordStore.openRecordStore("myDetails", true);

Closing a record store

void closeRecordStore()

myRcrds.closeRecordStore();

Deleting a record

deleteRecord(int recordId)

pass in the recordId for the record you would like to delete

myRecords.deleteRecord(1)

Deleting a record store

Pass the name of the record store.

RecordStore.deleteRecordStore(" myRcrds ");

Retrieving a record

Supply the method with the ID of the record you want

public byte[] getRecord(int recordId)throws RecordStoreNotOpenException

byte[] contactInfo = myRcrds.getRecord(1);

Replacing a record

This is similar to updating a record

setRecord(int recordId, byte[] newData, int offset, int numBytes)

- recordId - the ID of the record to use in this operation
- newData - the new data to store in the
- recordoffset - the index into the data buffer of the first relevant byte for this record

- numBytes - the number of bytes of the data buffer to use for this record

Interface RecordComparator

provides comparison between two records in a record store. It uses the compare method that takes two parameters i.e. the two records you would like to compare. It returns an integer value that corresponds to three preset values i.e. equivalent, precedes, and follows.

int compare(byte[] rec1, byte[] rec2)

Returns an integer that corresponds to the result

0 : EQUIVALENT

Two records are the same in terms of search or sort order

1 : FOLLOWS

rec1 follows rec2, rec1 comes after rec2

2 : PRECEDES

rec1 precedes rec2, rec1 comes before rec2

Example of Comparator RecordInterface.

```
/*-------------------------------------------------
* Compares two integers to determine sort order
* Each record passed in contains multiple Java data
* types - use only the integer data for sorting
*-------------------------------------------------*/
class ComparatorInteger implements RecordComparator
{
     private byte[] recData = new byte[10];

     // Read from a specified byte array
     private ByteArrayInputStream strmBytes = null;

     // Read Java data types from the above byte array
     private DataInputStream strmDataType = null;

     public void compareIntClose()
     {
       try
       {
         if (strmBytes != null)
           strmBytes.close();
         if (strmDataType != null)
```

```
      strmDataType.close();
    }
    catch (Exception e)
    {}
  }

  public int compare(byte[] rec1, byte[] rec2)
  {
    int x1, x2;

    try
    {
      // If either record is larger than our buffer, reallocate
      int maxsize = Math.max(rec1.length, rec2.length);
      if (maxsize > recData.length)
        recData = new byte[maxsize];

      // Read record #1
      // We want the integer from the record, which is
      // the second "field" thus we must read the
      // String first to get to the integer value
      strmBytes = new ByteArrayInputStream(rec1);
      strmDataType = new DataInputStream(strmBytes);
      strmDataType.readUTF();       // Read string
      x1 = strmDataType.readInt();  // Read integer

      // Read record #2
      strmBytes = new ByteArrayInputStream(rec2);
      strmDataType = new DataInputStream(strmBytes);
      strmDataType.readUTF();       // Read string
      x2 = strmDataType.readInt();  // Read integer

      // Compare record #1 and #2
      if (x1 == x2)
        return RecordComparator.EQUIVALENT;
      else if (x1 < x2)
        return RecordComparator.PRECEDES;
      else
        return RecordComparator.FOLLOWS;

    }
    catch (Exception e)
    {
      return RecordComparator.EQUIVALENT;
    }
  }
```

Adding a record

To add a new record, use the addRecord() method. The method takes 3 parameters:

data to be added. The data should be an array of bytes the offset. An integer value representing the starting position.

The length of the data to be written.

public int addRecord(byte[] data, int offset, int numBytes)

//create a buffer to hold/store our data

ByteArrayOutputStream bytedata=new ByteArrayOutputStream();

//create a data output stream that lets an application write primitive Java data types to an output stream

DataOutputStream data=new DataOutputStream(bytedata);

//write data into the output stream

data.writeUTF(content);

//use toByteArray to return the current contents of this output stream, as a byte array.

byte [] sdata=bytedata.toByteArray();

//add the record to the record store

record.addRecord(sdata,0,sdata.length);

//reset and close

bytedata.reset();

bytedata.close();

data.close();

Reading a record

//create a byte to hold the data read

byte[] recData = new byte[50000];

//Creates aByteArrayInputStream so that it uses recData as its buffer array. **ByteArrayInputStream input = new ByteArrayInputStream(recData);**

//create a datainputput stream

DataInputStream dInput = new DataInputStream(input);

for (int i = 1; i <= record.getNumRecords(); i++)

{

// use the getRecord method to retrieve the data. It takes three parameters

- **recordId** - the ID of the record to be retrieved
- **buffer** - the byte array in which to copy the data
- **offset** - the index into the buffer in which to start copying

record.getRecord(i, recData, 0);

//read and display the contents.

String finding = dInput.readUTF();

System.out.println("Data: " + finding);

Interface Record filter

Provide a mechanism to check if a record matches a specific criterion. RecordFilter has a single method, matches(), which is called for each record. Here's a simple

public boolean matches(byte[] candidate)

Record Listener

Provide an event listener that is notified when event is added, deleted or modified

Interface RecordEnumerator

The RecordStore's enumerateRecords()is used to enumerate records. Its basic function is to allow you to iterate through the records retrieved from the RecordStore. RecordEnumeration allows you to scroll both forward and backward through its contents. In addition, you can peek at the next or previous record ID.

It has the following methods:

- nextRecordId() – returns Id of next record.
- nextRecord() – returns next record.
- previousRecordId() – returns Id of previous record.
- previousRecord() – returns previous record.
- hasNextElement() – true, if there is next record.

Revision Exercise

1. Explain the functions of RecordEnumerator interface.
2. Differentiate between recordComparator and recordListener.
3. Explain how Record Management System works in MIDlet Application.
4. Write a simple syntax for read record method of MIDlet Application.
5. Write a MIDlet application with the following functionalities.
   i. **Add record**
   ii. **Delete record**
   iii. **Replace record**
6. Differentiate between getRecord() and getNumRecord() methods of RMS.

7. The Generic Connection Framework

Objectives:

At the end of this topic the students should be able to:

- Understand Generic Connect Framework.
- Understand Connection Interfaces
- Create MIDlet GUI application that uses GCF.

Introduction

The Generic Connection framework is introduced in J2ME's CLDC to reflect the requirements of small-footprint networking and file I/O for a broad range of mobile devices.

To meet the small-footprint requirement, the Generic Connection framework generalizes the functionality of J2SE's network and file I/O classes from J2SE's java.io and jave.net packages. It is a precise functional subset of J2SE classes, but much smaller in size. These J2ME classes and interfaces are all included in a single package, javax.microedition.io.

To meet the extendibility and flexibility requirement, the Generic Connection framework uses a set of related abstractions for different forms of communications, represented by seven connection interfaces:

i. **Connection.**

ii. **ContentConnection.**

iii. **DatagramConnection.**

iv. **InputConnection.**

v. **OutputConnection.**

vi. **StreamConnection.**

vii. StreamConnectionNotifier.

The GCF has the interfaces and classes to create connections such as HttpConnection, HttpsConnection and to perform the IO operations.

GCF has the following interfaces:

i. **Connection-**The most basic type of connection in the GCF. All other connection types extend Connection.
ii. **ContentConnection-**Manages a connection, such as HTTP, for passing content, such as HTML or XML. Provides basic methods for inspecting the content length, encoding and type of content.
iii. **Datagram**-Acts as a container for the data passed on a Datagram Connection.
iv. **DatagramConnection**-Manages a datagram connection.
v. **InputConnection**-Manages an input stream-based connection.
vi. **OutputConnection**-Manages an output stream-based connection.
vii. **StreamConnection**-Manages the capabilities of a stream. Combines the methods of both InputConnection and OutputConnection.
viii. **StreamConnectionNotifier**-Listens to a specific port and creates a StreamConnection as soon as activity on the port is detected.
ix. **Connector**-The Connector class is used to create instances of a connection protocol using one of Connector's static methods. The Connector class is not designed to be instantiated. It is simply used to create instances of a connection protocol. All of the methods Connector defines are static and serve this purpose. The Connector defines three variations of an open() that return a Connection instance.

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

The name is essentially a URI and is composed of three parts: a scheme, an address, and a parameter list. The general form of the name parameter is as follows: <scheme>:<address>;<parameters> The mode parameter allows the connection to be established in various access modes, such as read-only, read-write and write-only. These modes are defined

by the Connector constants READ, READ_WRITE, and WRITE. The timeouts parameter is a flag indicating whether or not the connection should throw an InterruptedIOException if a timeout occurs.

The Generic Connection framework supports the following basic forms of communications. All the connections are created by one common method, **Connector.open():**

i. HTTP:

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

ii. Sockets:

```
Connector.open("socket://localhost:80");
```

iii. Datagrams:

```
Connector.open("datagram://http://www.webyu.com:9000");
```

iv. Serial Port:

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

v. File:

```
Connector.open("file:/foo.dat");
```

Connection Interfaces

`Connection` is the base interface, the root of the connection interface hierarchy. All the other connection interfaces derive from `Connection`. `StreamConnection` derives from `InputConnection` and `OutputConnection`. It defines both the input and output capabilities for a stream connection. `ContentConnection` derives from `StreamConnection`. It adds three methods for MIME data handling to the input and output methods in `StreamConnection`. Finally, `HttpConnection` derives from `ContentConnection`.

The `HttpConnection` is not part of the Generic Connection framework. Instead, it is defined in the MIDP specification that is targeted at cellular phones and two-way pagers. `HttpConnection` contains methods and constants specifically to support the HTTP 1.1 protocol. The `HttpConnection` interface must be implemented by all MIDP implementations, which means the

`HttpConnection` will be a concrete class in the actual MIDP implementations. The http communication capability is expected to be available on all MIDP devices.

These connections are defined as interfaces instead of concrete classes since the Generic Connection framework specifies only the basic framework of how these connection interfaces should be implemented. The actual implementations are left to individual device manufacturers' profile implementations (MIDP or PDAP).Individual device manufacturers may be interested in implementing only a subset of these connection interfaces based on the capabilities of their devices. The documentation of each manufacturer's J2ME MIDP SDK should indicate which connection interfaces are implemented and which are not.

Creating Network Connections

The Connector class is the core of the Generic Connection framework, because all connections are created by the static open() method in the Connector class. Different types of communication are created from the same method. The connection type can be file I/O, serial port communication, datagram connection, or an HTTP connection, depending on the string parameter passed to the method. Such a design makes the J2ME implementation more extensible and flexible in supporting new devices and product.

The method's signature is as follows:

Connection open(String connect_string)

The connect_string has a format of {protocol}:[{target}][{params}], which is similar to the commonly used URL format, such as http://www.nowhere.com. It consists of three parts: **protocol, target**, and **params**.

protocol dictates what type of connection will be created by the **open()** method. There are several possible values for protocol, as shown in the table below.

Protocol Values

Value	Connection Usage

Value	Connection Usage
File	File I/O
Comm.	Serial port communication
Socket	TCP/IP socket communication
Datagram	Datagram communication
http	Accessing Web servers

Table 6.1

`target` can be a hostname, a network port number, a file name, or a communication port number.

`params` is optional. It specifies the additional information needed to complete the connect string.

The following examples demonstrate how to use the `open()` method to create different types of communication based on different protocols:

HTTP communication:

```
Connection hc = Connector.open("http://www.nowhere.com")
```

Socket communication:

```
Connection sc = Connector.open("socket://localhost:9000")
```

Datagram communication:

```
Connection dc =
Connector.open("datagram://http://www.nowhere.com:9000")
```

Serial port communication:

```
Connection cc = Connector.open("comm:0;somerate=9000")
```

File I/O:

```
Connection fc = Connector.open("file:/myfile.txt")
```

Actual support for these protocols varies from vendor to vendor. MotoSDK supports all three networking protocols that are: **socket (TCP/IP), datagram (UDP), and http (HTTP).** Most of l the sample codes in this session are compiled and run under MotoSDK.

A ConnectionNotFoundException will be thrown if your MIDlet program tries to create a connection based on a protocol that is not supported by your device manufacturer's MIDP implementation.

The Methods in the Connector Class

The Connector class is the only concrete class in Generic Connection framework in CLDC. It contains seven static methods:

1. **static Connection open(String connectString)**

This method creates and opens a new Connection based on the connectString.

2. **static Connection open(String connectString, int mode)**

This method creates and opens a new Connection based on the connectString. The additional mode parameter specifies the access mode for the connection. There are three access modes:

- Connector.READ
- Connector.READ_WRITE.
- Connector.WRITE.

If mode is not specified, the default value is Connector.READ_WRITE. The validity of the actual setting is protocol dependent. If the access mode is not allowed for a protocol, an IllegalArgumentException will be thrown.

3. **static Connection open(String connectString, int mode, boolean timeouts)**

This method creates and opens a new Connection based on the connectString. The additional timeouts parameter is a Boolean flag that dictates whether the method will throw a timeout exception InterruptedIOException. The default timeouts value is false, which indicates that no exception will be thrown.

4. **static DataInputStream openDataInputStream(String connectString)**

This method creates and opens a new DataInputStream based on the connectString.

5. **static DataOutputStream openDataOutputStream(String connectString)**

This method creates and opens a DataOutputStream from the connectString.

6. **static InputStream openInputStream(String connectString)**

This method creates and opens a new InputStream from the connectString.

7. **static OutputStream openOutputStream(String connectString)**

This method creates and opens a new OutputStream from the connectString.

The last four I/O stream-creation methods combine creating the connection and opening the input/output stream into one step. For example, the following statement

DataInputStream dis = Connector.openDataInputStream(http://www.nowhere.com);

is the equivalent of the following two statements:

```
InputConnection ic = (InputConnection)
 Connector.open("http://www.nowhere.com", Connector.READ, false);
DataInputStream dis = ic.openDataInputStream();
```

An IllegalArgumentException will be thrown if a malformed connectString is received. A ConnectionNotFoundException will be thrown if the protocol specified in connectString is not supported. An IOException will be thrown for other types of I/O errors.

Example 7-1 contains an example of how an http connection is created and how a DataInputStream is opened on top of the connection.

Example 7-1. tester.txt

```
/**
 * This sample code block demonstrates how to open an
 * http connection, how to establish an InputStream from
 * this http connection, and how to free them up after use.
 **/
// include the networking class libraries
import javax.microedition.io.*;
// include the I/O class libraries
import java.io.*;
// more code here ...
// define the connect string with protocol: http
// and hostname: 192.168.2.1
String connectString = "http://192.168.2.1";
InputConnection hc = null;
DataInputStream dis = null;
// IOException must be caught when Connector.open() is called
try {
  // an http connection is established with read access.
  // The returned object is cast into an InputConnection object.
  hc = (InputConnection)
   Connector.open(connectString, Connector.READ, false);
  // an InputStream is created on top of the InputConnection
  // object for read operations.
  dis = hc.openDataInputStream();
  // perform read operations here ...
} catch (IOException e) {
  System.err.println("IOException:" + e);
} finally {
```

```
  // free up the I/O stream after use
  try { if (dis != null ) dis.close(); }
  catch (IOException ignored) {}

  // free up the connection after use
  try { if ( hc != null ) hc.close(); }
  catch (IOException ignored) {}
}
// more code here ...
```

Connection Interfaces

The connection interfaces defined in the javax.microedition.io package including the seven connections from the Generic Connection framework in CLDC and the HttpConnection in MIDP:

Connection
ContentConnection
DatagramConnection
InputConnection
OutputConnection
StreamConnection
StreamConnectionNotifier
HttpConnection

Connection

The Connection interface has one method:

```
void close()
```

This method closes the connection.

InputConnection

The InputConnection interface has two methods:

DataInputStream openDataInputStream()

This method opens a data input stream from the connection.

InputStream openInputStream()

This method opens an input stream from the connection.

OutputConnection

The OutputConnection interface has two methods:

DataOutputStream openDataOutputStream()

This method opens a data output stream from the connection.

OutputStream openOutputStream()

This method opens an output stream from the connection.

DatagramConnection

The DatagramConnection interface is used to create a datagram for a UDP communication. This interface has eight methods:

int getMaximumLength()

This method returns the maximum length that is allowed for a datagram packet.

int getNominalLength()

This method returns the nominal length for datagram packets.

Datagram newDatagram(byte[] buf, int size)

This method creates a new Datagram object. buf is the placeholder for the data packet, and size is the length of the buffer to be allocated for the Datagram object.

Datagram newDatagram(byte[] buf, int size, String addr)

This method creates a new Datagram object. The additional parameter addr specifies the destination of this datagram message. It is in the format {protocol}://[{host}]:{port}.

Datagram newDatagram(int size)

This method creates a new Datagram object with an automatically allocated buffer with length size.

Datagram newDatagram(int size, String addr)

This method creates a new Datagram object. addr specifies the destination of this datagram message.

void receive(Datagram dgram)

This method receives a Datagram object dgram from the remote host.

void send(Datagram dgram)

This method sends a Datagram object dgram to the remote host.

StreamConnection

The StreamConnection interface offers both send and receive capabilities for socket-based communication. All four of its methods are inherited from InputConnection and OutputConnection:

- DataInputStream openDataInputStream()
- InputStream openInputStream()
- DataOutputStream openDataOutputStream()
- OutputStream openOutputStream()

StreamConnectionNotifier

The StreamConnectionNotifier interface has one method:

StreamConnection acceptAndOpen()

This method returns a StreamConnection that represents a server-side socket connection to communicate with a client.

ContentConnection

The ContentConnection interface extends from StreamConnection and adds three methods to determine an HTTP stream's character encoding, MIME type, and size:

String getEncoding()

This method returns the value of the content-encoding in the HTTP header of an HTTP stream.

long getLength()

This method returns the value of the content-length in the HTTP header of an HTTP stream.

String getType()

This method returns the value of the content-type in the HTTP header of an HTTP stream.

The methods inherited from StreamConnection are as follows:

- **DataInputStream openDataInputStream()**
- **InputStream openInputStream()**
- **DataOutputStream openDataOutputStream()**
- **OutputStream openOutputStream()**

HttpConnection

The HttpConnection extends from ContentConnection. It adds the following methods to support the HTTP 1.1 protocol:

- long getDate()
- long getExpiration()
- String getFile()
- String getHeaderField(int index)
- String getHeaderField(String name)
- long getHeaderFieldDate(String name, long def)
- int getHeaderFieldInt(String name, int def)
- String getHeaderFieldKey(int n)
- String getHost()
- long getLastModified()
- int getPort()
- String getProtocol()
- String getQuery()
- String getRef()
- String getRequestMethod()
- String getRequestProperty(String key)
- int getResponseCode()
- String getResponseMessage()
- String getURL()
- void setRequestMethod(String method)
- void setRequestProperty(String key, String value)

The HttpConnection interface is mandatory for all MIDP vendor implementations. However, the underlying support mechanisms could be different from vendor to vendor. Some vendors may support the HTTP stack on top of non-IP–based protocols such as WSP transport or TL/PDC-P, and other vendors may use HTTP over TCP/IP. To support the HTTP protocol on MIDP devices, non-IP networks may have to install gateways in order to convert HTTP requests from the wireless network format to a TCP/IP format to be able to access the Internet.

Revision Exercise.

1. Name any three connection interfaces in J2ME applications.
2. Discuss the Generic Connection Framework in J2ME.
3. Differentiate between DataInputStream() and DataInput() methods.
4. Name three access modes of **static Connection open(String connectString, int mode)**
5. Write a simple code in MIDlet application to open and establish an HTTP connection.
6. Write a simple code that uses the DatagramConnection interface.

8. HTTP/HTTPS Connection in J2ME.

Objectives:

At the end of this topic the students should be able to:

- Understand HTTP connections.
- Understand HTTPs connections
- Create MIDlet GUI application that uses both HTTP and HTTPS connection.

HttpConnection

HTTP is the most popular data protocol of the Internet. Most wireless data networks support HTTP on top of their native wireless data carrier layers. The MIDP specification mandates support for HTTP version 1.1 on all Java handsets. We can use HTTP connections to pass both text and binary data between clients and servers.

```
HttpConnection hc = (HttpConnection)
Connector.open("http://www.somewhere.intheworld.com/Community/Wiki/");
```

HTTP data is transferred over continuous streams. Hence, the HttpConnection inherits from the StreamConnection interface. The two most important methods in HttpConnection open data streams to and from the server.

DataInputStream getDataInputStream ()

DataOutputStream getDataOutputStream ()

The HttpConnection interface supports both HTTP GET or POST operations. In a GET operation, data is retrieved from the specified URL. We can customize the URL to pass request arguments. In this case, we only need to call the **openDataInputStream()** method to make the physical connection and read the data from the DataInputStream.

In a POST operation, we have to open a DataOutputStream to the specified URL first and write the request parameters to the stream. A connection is made, and the request is submitted when

we close the DataOutputStream. Then we open a DataInputStream to retrieve the server response.

```
HttpConnection conn =

(HttpConnection)Connector.open("http://www.somewhere.intheworld.com/Community/Wiki/");
conn.setRequestMethod(HttpConnection.GET);
DataInputStream din = conn.openDataInputStream();
ByteArrayOutputStream bas = new ByteArrayOutputStream();
byte[] buf = new byte[256];
while (true) {
int rd = din.read(buf, 0, 256);
  if (rd == -1) break;
  bos.write(buf, 0, rd);
  }
bas.flush();
buf = bos.toByteArray();
//Convert the byte array to string using the
//character encoding format of the remote site
String content = new String (buf, "UTF-8");
```

HttpsConnection

The HttpsConnection object represents a secure HTTP (HTTPS) connection. HTTPS support is mandated in MIDP 2.0 . It supports both user authentication and data encryption. The URL scheme for HttpsConnection is https://. SomepartoftheURL:

```
HttpsConnection hsc =
(HttpsConnection) Connector.open("https://secureserver.com");
```

The HTTPS protocol utilizes the public key infrastructure to authenticate communication parties and ensure data confidentiality once the connection is established. Digital certificates are used for authentication, and public key algorithms are used to exchange the information

1. **SocketConnection**-A socket connection allows the client to communicate to specific socket ports. The URL string of SocketConnection objects takes the format of **socket://host:port**(discussed this ealier).

The important methods in the SocketConnection interface are:

i. **openDataInputStream().**
ii. **openDataOutputStream(),**

which open data streams for the client to communicate with the remote socket.

We can also specify the operation mode of the SocketConnection using the setSocketOption() method. void setSocketOption (byte option, int value) int getSocketOption (byte option).

2. **SecureConnection** A SecureConnection interface extends the SocketConnection interface to add support for the Secure Socket Layer (SSL). Its URL string takes the form **ssl://host:port**
3. **ServerSocketConnection** The ServerSocketConnection allows us to listen for incoming connections on a socket port and essentially turns the device into a server this is not available in MIDP 1.0.

   Its URL string has the format of **socket://:port**, where the port parameter is the port number to listen to. An external device can now connect to the server device using the **socket://host:port** connection string, where host is the IP address of the server device. Once a **ServerSocketConnection** object is instantiated, we can call the **acceptAndOpen()** method in a loop to listen for incoming connections.

```
// Create the server listening socket for port 90

ServerSocketConnection scn =
(ServerSocketConnection) Connector.open("socket://:90");
while (true) {
  // Wait for a connection
  SocketConnection sc =
```

```
    (SocketConnection) scn.acceptAndOpen();
  sc.setSocketOption(RCVBUF, 128);
  sc.setSocketOption(SNDBUF, 128);
  // do something with sc
 sc.close ();
    }
scn.close ();
```

4. **CommConnection**-This supports access to the device's serial port or logical serial port (e.g., the IrDA port). The URL scheme takes the form comm:port_id;params. The port_id is COM# for RS232 serial ports and IR# for IrDA ports. For example, the URL string comm:IR0;baudrate=19200 opens a connection to the first infrared port with a 19,200 bit per second data rate.
5. **UDPDatagramConnection** The datagram protocol directly passes data packets between endpoints without trying to assemble them to match their originating order or recover lost data. The datagram arrival order or even the delivery itself is not guaranteed. It is, however, often faster and more suitable for real-time applications.

   The UDPDatagramConnection interface inherits from the DatagramConnection interface. The URL string is datagram**://host:port.** It cannot open data streams but has methods to support composing, sending, and receiving datagrams.

Datagram newDatagram(int i)
Datagram newDatagram(int i, String s)
Datagram newDatagram(byte[] bytes, int i)
Datagram newDatagram(byte[] bytes, int i, String s)
void send(Datagram datagram)
void receive(Datagram datagram)

```
/*
* Network Example
*/
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
```

```
import javax.microedition.io.*;
import java.util.*;
import java.io.*;

public class MyNetEx extends MIDlet {

private static Command httpTestCommand = new Command("HttpConn",
Command.SCREEN, 1);
private static Command datagramSendCommand = new Command("Send Datagram",
Command.SCREEN, 1);
private static Command datagramReceiveCommand = new Command("Receive
Datagram", Command.SCREEN, 1);
private static Command socketClientCommand = new Command("Socket Client",
Command.SCREEN, 1);
private static Command socketServerCommand = new Command("Socket Server",
Command.SCREEN, 1);
private Command commandType;

private List list = new List("Http Test", List.IMPLICIT);

/**
Used to create instances of the MIDlet
*/
public MyNetEx()
{
CommandListener listener = initListeners();
Display.getDisplay(this).setCurrent(list);
list.addCommand(httpTestCommand);
list.addCommand(datagramSendCommand);
list.addCommand(datagramReceiveCommand);
list.addCommand(socketClientCommand);
list.addCommand(socketServerCommand);
```

```
list.setCommandListener(listener);
}
 /**
Used to create instances of Thread to run the actual command
*/
public MyNetEx(Command cmd)
{
commandType = cmd;
}
 public void startApp()
{
}
 public void pauseApp()
{
}
 public void destroyApp(boolean unc)
{
}
 private void appendData(String data)
{
list.append(data, null);
}
 /**
Create a listener command handler
*/
private CommandListener initListeners()
{
return new CommandListener()
{
public void commandAction(Command command, Displayable displayable)
{
Thread t = new Thread(new CommandHandler(command));
```

```
t.start();
}
};
}

class CommandHandler implements Runnable
{
private Command commandType;

public CommandHandler(Command type)
{
commandType = type;
}
 private void httpTest()
{
 HttpConnection c = null;
InputStream is = null;
StringBuffer sb = new StringBuffer();
try
{
c = (HttpConnection)Connector.open("<nowhere>http://localhost</nowiki>",
Connector.READ_WRITE, true);
c.setRequestMethod(HttpConnection.GET); //default
is = c.openInputStream(); // transition to connected!
int ch = 0;
for(int ccnt=0; ccnt &lt; 150; ccnt++) //get the title.
{
ch = is.read();
System.out.print((char)ch);
if (ch == -1)
{<br>break;<br>}<br>sb.append((char)ch);
}<br>}<br>catch (IOException x)
```

```
{<br>x.printStackTrace();
}
finally
{
try
{
is.close();
c.close();
} catch (IOException x)
{
x.printStackTrace();
}
}
appendData(sb.toString());
}

private void sendDatagram()
{
try
{
DatagramConnection dgc = (DatagramConnection)
Connector.open("datagram://localhost:8080");
try
{
byte[] payload = "Hello from a Datagram".getBytes();<br>Datagram datagram =
dgc.newDatagram(payload, payload.length);
dgc.send(datagram);
} finally
{
dgc.close();
}
} catch (IOException x)
```

```
{
x.printStackTrace();
}
}

private void receiveDatagram()
{
try
{
DatagramConnection dgc = (DatagramConnection)
Connector.open("datagram://:8080");
try
{
int size = 200;
Datagram datagram = dgc.newDatagram(size);
dgc.receive(datagram);
appendData(new String(datagram.getData()).trim());
} finally
{
dgc.close();
}
} catch (IOException x)
{
x.printStackTrace();
}
}
 private void clientSocket()
{
try
{
SocketConnection sc = (SocketConnection)
Connector.open("socket://localhost:8081");<br>OutputStream os =
```

```
null;<br>try<br>{<br>os = sc.openOutputStream();
byte[] data = "Hello from a socket!".getBytes();
os.write(data);
} finally
{
sc.close();
os.close();
}
} catch (IOException x)
{
x.printStackTrace();
}
}
throw new IllegalStateException("No command specified to run.");
}
}
```

Revision Exercise.

1. Explain the difference between HTTP and HTTPS connections.
2. Write a simple code snippet in MIDlet application that uses POST operation to establish a HttpConnection to the server of your choice.

9. TCP Sockets Programming in J2ME

Objectives:

At the end of this topic the students should be able to:

- Understand TCP sockets.
- Understand the difference between Sockets and Datagrams.
- Understand Wireless Network Programming Using Sockets.
- Create a simple J2ME application that uses sockets.

Intoduction

Sockets are different from datagram since they use a connection-based paradigm to transmit data. That's both a sender and a receiver must be running and establish a communication channel for data to be exchanged. To use a real-world analogy, a socket connection is like calling a friend on the telephone. If the friend does not answer, a conversation cannot take place. Datagrams on the other hand are more like sending a letter to a friend, where a note is placed into an envelope, addressed, and mailed.

The following code demonstrates how to set up a listener to monitor a port for an inbound socket connection.

```
try
{
  ServerSocketConnection ssc = (ServerSocketConnection)
  Connector.open("socket://:9002");
  StreamConnection sc = null;
  InputStream is = null;
  try{
    sc = ssc.acceptAndOpen();
    is = sc.openInputStream();
    int ch = 0;
    StringBuffer sb = new StringBuffer();
    while ((ch = is.read()) != -1){
```

```
      sb.append((char)ch);
    }
    System.out.println(sb.toString());
  } finally{
      ssc.close();
      sc.close();
      is.close();
  }
} catch (IOException x) {
    x.printStackTrace();
}
```

In this example a ServerSocketConnection is opened on port 9002. This type of connection is used for sole purpose of listening for an inbound socket connection. The code goes into a wait state when the acceptAndOpen() method is called. When a socket connection is established, the acceptAndOpen() method returns with an instance of a SocketConnection. Opening an input stream on this connection allows data to be read from the transmission.

The next example demonstrates the code required by the client to initiate the socket connection.

```
try{
  SocketConnection sc = (SocketConnection)
    Connector.open("socket://localhost:9002");
  OutputStream os = null;
  try{
    os = sc.openOutputStream();
    byte[] data = "Hello from a socket!".getBytes();
    os.write(data);
  } finally{
      sc.close();
      os.close();
  }
```

```
} catch (IOException x){
        x.printStackTrace();
}
```

In this example a SocketConnection is established on port 9002 of the local machine. When using sockets, this is the point on the server side that the acceptAndOpen() method returns. If the connection is successfully opened, the OutputStream is obtained and a message is written to the stream. Note that because sockets are connection based, if there is no server listening for an incoming socket connection an exception will be thrown.

Wireless Network Programming Using Sockets

A *socket* is one end-point of a two-way communication link between programs running on the network. A socket connection is the most basic low-level reliable communication mechanism between a wireless device and a remote server or between two wireless devices. The socket communication capability provided with some of the mobile devices in J2ME enables a variety of client/server applications. Using the socket connection to SMTP and POP3 servers, an e-mail client on wireless devices can send or receive regular Internet e-mail messages.

Socket use gives J2ME developers the flexibility to develop all kinds of network applications for wireless devices. However, not every wireless manufacturer supports socket communication in MIDP devices, which means that wireless applications developed using sockets could be limited to certain wireless devices and are less likely to be portable across different types of wireless networks.

To use a socket, the sender and receiver that are communicating must first establish a connection between their sockets. One will be listening for a request for a connection, and the other will be asking for a connection. Once two sockets have been connected, they may be used for transmitting data in either direction.

To receive data from the remote server, InputConnection has to be established and an InputStream must be obtained from the connection. To send data to the remote server, OutputConnection has to be established and an OutputStream must be obtained from the connection. In J2ME, three types of connections are defined to handle input/output streams:

- InputConnection,
- OutputConnection,
- StreamConnection.

InputConnection defines the capabilities for input streams to receive data.

OutputConnection defines the capabilities for output streams to send data.

StreamConnection defines the capabilities for both input and output streams. When to use which connection depends on whether the data needs to be sent, received, or both.

Network programming using sockets is very straightforward in J2ME. The process works as follows:

1. A socket connection is opened with a remote server or another wireless device using Connector.open().
2. InputStream or OutputStream is created from the socket connection for sending or receiving data packets.
3. Data can be sent to and received from the remote server via the socket connection by performing read or write operations on the InputStream or OutputStream object.
4. The socket connection and input or output streams must be closed before exiting the program.

The example below demonstrates how a socket connection is created and how a DataOutputStream is opened on top of the connection.

```
/**
 * This sample code block demonstrates how to open a
 * socket connection, how to establish a DataOutputStream
 * from this socket connection, and how to free them up
 * after use.
 **/
// include the networking class libraries
```

```
import javax.microedition.io.*;
// include the I/O class libraries
import java.io.*;

// more code here ...

// define the connect string with protocol: socket,
// hostname: 192.168.2.1, and port number: 80
String connectString = "socket:// 192.168.2.1";
OutputConnection sc = null;
DataOutputStream dos = null;

// IOException must be caught when Connector.open() is called.
try {
  // a socket connection is established with the remote server.
  sc = (OutputConnection) Connector.open(socketUrlString);

  // an OutputStream is created on top of the OutputConnection
  // object for write operations.
  dos = sc.openDataOutputStream();

  // perform write operations that send data to the remote server ...

} catch (IOException e) {
  System.err.println("IOException caught:" + e)
} finally {
  // free up the I/O stream after use
  try { if (dos != null ) dos.close(); }
  catch (IOException ignored) {}

  // free up the socket connection after use
  try { if ( sc != null ) sc.close(); }
```

```
  catch (IOException ignored) {}
}

// more code here ...
```

From this example, Socket connection is established with remote server **192.168.2.1** on port 80. Port 80 is the well-known port for HTTP service. Note that we are using the socket connection to communicate with the remote server; therefore, the protocol is specified as socket. The DataOutputStream is then created on the top of the connection for sending requests to the remote server.

Program Example

The sample program below creates a Web client session to request a page from a Web server on the Internet illustrates the program flow of a Web client implemented using socket connections.

SocketExample.java

```
/**
 * The following MIDlet application creates socket connection with
 * a remote Web server at port 80, and then sends an HTTP request
 * to retrieve the Web page "index.html" via the connection.
 */

// include MIDlet class libraries
import javax.microedition.midlet.*;
// include networking class libraries
import javax.microedition.io.*;
// include GUI class libraries
import javax.microedition.lcdui.*;
// include I/O class libraries
import java.io.*;

public class SocketExample extends MIDlet {
```

```
// StreamConnection allows bidirectional communication
private StreamConnection streamConnection = null;

// use OutputStream to send requests
private OutputStream outputStream = null;
private DataOutputStream dataOutputStream = null;

// use InputStream to receive responses from Web server
private InputStream inputStream = null;
private DataInputStream dataInputStream = null;

// specify the connect string
private String connectString = "socket:// 192.168.2.1";

// use a StrignBuffer to store the retrieved page contents
private StringBuffer results;

// define GUI components
private Display myDisplay = null;
private Form resultScreen;
private StringItem resultField;

public SocketExample() {
  // initializing GUI display
  results = new StringBuffer();
  myDisplay = Display.getDisplay(this);
  resultScreen = new Form("Page Content:");
}

public void startApp() {
  try {
    // establish a socket connection with remote server
```

```
  Code to create connection here
  // create DataOuputStream on top of the socket connection
  outputStream = streamConnection.openOutputStream();
  dataOutputStream = new DataOutputStream(outputStream);

  // send the HTTP request
  Some codes

  // create DataInputStream on top of the socket connection
  inputStream = streamConnection.openInputStream();
  dataInputStream = new DataInputStream(inputStream);

  // retrieve the contents of the requested page from Web server
  int inputChar;
  while ( (inputChar = dataInputStream.read()) != -1) {
    results.append((char) inputChar);
  }

  // display the page contents on the phone screen
  resultField = new StringItem(null, results.toString());
  resultScreen.append(resultField);
  myDisplay.setCurrent(resultScreen);

} catch (IOException e) {
  System.err.println("Exception caught:" + e);
} finally {
  // free up I/O streams and close the socket connection
  try {
    if (dataInputStream != null)
      dataInputStream.close();
  } catch (Exception ignored) {}
  try {
```

```
      if (dataOutputStream != null)
        dataOutputStream.close();
    } catch (Exception ignored) {}
    try {
      if (outputStream != null)
        outputStream.close();
    } catch (Exception ignored) {}
    try {
      if (inputStream != null)
        inputStream.close();
    } catch (Exception ignored) {}
    try {
      if (streamConnection != null)
        streamConnection.close();
    } catch (Exception ignored) {}
  }
 }

 public void pauseApp() {
 }

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

The program first opens a socket connection with the Web server http://www.nowhere.com at port 80. It sends an HTTP request to the Web server using the DataOutputStream established from the connection. It receives the requested content from the Web server using the DataInputStream opened from the connection. After the Web page content is completely received, the content is displayed on the emulator.Because the program needs to perform bidirectional communication between a cell phone and a Web server, StreamConnection is used to support both the send and receive operations.

Revision Exercises.

1. Discuss the TCP sockets programming in J2ME applications.
2. Explain the difference between Sockets and Datagrams.
3. Discuss Wireless Network Programming Using Sockets.
4. Create a simple J2ME application that uses sockets.

10. J2ME Optional Packages

Objectives:

At the end of this topic the students should be able to:

- Understand Optional packages in J2ME application.
- Understand the difference J2ME optional packages.
- Create a simple J2ME application that uses optional packages supported by J2ME.

When the Java 2 Platform, Micro Edition (J2ME) was first introduced, only one configuration, the Connected Limited Device Configuration (CLDC), and one profile, the Mobile Information Device Profile (MIDP) had been defined as formal specifications through the Java Community Process (JCP). Today, there are nearly forty J2ME-related specifications at various stages in the JCP, and many of these specifications define optional packages instead of configurations or profiles.

The J2ME Platform

Initially, the J2ME platform consisted only of configurations and profiles. A *configuration* defines the minimum Java runtime environment for a family of devices: the combination of a Java virtual machine (either the standard J2SE virtual machine or a much more limited version called the CLDC VM) and a core set of application programming interfaces (APIs).

A ***profile*** is a set of APIs added to a configuration to support specific uses of a device. Profiles define complete, and usually self-contained, application environments. Profiles often, but not always, define user interface and persistence APIs; the MIDP fits this pattern. Profiles may be supersets or subsets of other profiles; the Personal Basis Profile is a subset of the Personal Profile and a superset of the Foundation Profile.

The overall architecture of the J2ME platform needed more precise definition i.e using the optional package. A Java Specification Request (JSR) formally defining the platform was created. The J2ME Platform Specification (also known by its specification number, JSR 68),

defines J2ME configurations, profiles, and optional packages in terms of lower-level elements. JSR 68 is currently in the community review phase of the JCP. The reuse of existing APIs, whether those APIs are defined by J2ME, J2SE, or J2EE, is strongly encouraged whenever possible. Reuse prevents unnecessary fragmentation of the Java platform as a whole and promotes code portability across its three editions.

Optional Package

An optional package is also a set of APIs, but unlike a profile, it does not define a complete application environment. An optional package is always used in conjunction with a configuration or a profile. It extends the runtime environment to support device capabilities that are not universal enough to be defined as part of a profile or that need to be shared by different profiles.

Consider the Wireless Messaging API (WMA), a set of classes for sending and receiving Short Message Service (SMS) messages. Because the WMA is an optional package, it can be included on any J2ME device with SMS capabilities, not just MIDP-enabled cellphones. If WMA were part of a specific profile, such as MIDP, its use would have been limited to that profile and its supersets.

Because, just like profiles and configurations, optional packages are specified through the Java Community Process, each has its own reference implementation (RI) and test compatibility toolkit (TCK).

Using Optional Packages

For the application developer, an optional package is just another set of Java classes to place in the Java compiler's classpath. The classes are not packaged with the applications, of course, because the devices that support the optional package include them in their runtime environments.

To detect the presence or absence of an optional package, test for the existence of a class unique to the optional package:

```
...
public static boolean isWMAPresent(){
    try {
        Class.forName(
               "javax.wireless.messaging.MessageConnection" );
        return true;
    }
    catch( Exception e ){
        return false;
    }
}
...
```

Choose the class to test with care. For example, the existence of the `java.rmi.Remote` interface does not imply that the RMI Optional Package (JSR 66) is supported, because the Personal Basis Profile includes the same interface as part of its support for inter-Xlet communication.

Example Optional Packages

Several optional packages have already been defined through the Java Community Process, and more are under development. Here are brief summaries of the more interesting and important optional packages:

1. JSR 66: RMI Optional Package

Remote method invocation (RMI), a feature of J2SE, enables Java objects running in one virtual machine to invoke methods of Java objects running in another virtual machine seamlessly. The RMI Optional Package (RMIOP) extends this capability to the J2ME platform. The RMIOP is a subset of J2SE 1.3's RMI functionality. It is for use in CDC-based profiles that incorporate the Foundation Profile, such as the Personal Basis Profile and the Personal Profile. The RMIOP *cannot* be used with CLDC-based profiles because they lack object serialization and other important features found only in CDC-based profiles.

The RMIOP supports most of the J2SE RMI functionality, including the Java Remote Method Protocol, marshalled objects, distributed garbage collection, registry-based object lookup, and network class loading. HTTP tunneling and the Java 1.1 stub protocol are not supported.

Using the RMIOP is mostly a matter of avoiding the unsupported classes and features of the full RMI specification. You can use the standard rmic tool from the J2SE Software Development Kit to generate the RMI stub classes.

2. JSR 120: Wireless Messaging API

The Wireless Messaging API lets J2ME applications send and receive messages using the Short Message Service, a messaging protocol used by cellphones across the world. The WMA extends the CLDC's Generic Connection Framework (GCF) by defining a new connection interface, **javax.wireless.messaging.MessageConnection,** and exposing SMS and CBS connections (short for Cell Broadcast Service, a related protocol) through protocol handlers for URLs beginning with " sms:" or " cbs:". For example, here's how to send an SMS text message with the WMA:

...

```
import javax.microedition.io.*;
import javax.wireless.messaging.*;
...
MessageConnection conn = null;
String url = "sms://+254789002221";

try {
   conn = (MessageConnection) Connector.open( url );
   TextMessage msg = conn.newMessage( conn.TEXT_MESSAGE );
   msg.setPayloadText( "Please call me!" );
   conn.send( msg );
}
catch( Exception e ){
```

```
   // handle errors
}
finally
        {
   if( conn != null ){
      try { conn.close();
           } catch( Exception e ){}
   }
}
...
```

The WMA can be used with any J2ME profile.

3. JSR 135: Mobile Media API

The Mobile Media API (MMAPI) defines abstractions for the capture and playback of multimedia content. If a device supports the playback of specific audio or video formats, or the recording of audio or video content (such as through an integrated camera), those capabilities can be exposed generically through the MMAPI. Downloading and playing a music recording, for example, is as simple as doing this:

```
...

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

...

try {
   Player p = Manager.createPlayer(
              "http://somesite.com/music.mp3" );
```

```
    p.start();
}
catch( IOException ioe ){
}
catch( MediaException me ){
}
...
```

The code above assumes, of course, that the device's capabilities include playing MP3-encoded streams. The full MMAPI can be used with any J2ME profile. A subset of the MMAPI has actually been incorporated into the Mobile Information Device Profile 2.0.

4. JSR 169: JDBC for CDC/FP

Java applications communicate with relational database servers using JDBC, a feature of J2SE. The JDBC for CDC/FP optional package (JDBCOP) is a strict subset of JDBC 3.0 that excludes some of JDBC's advanced and server-oriented features, such as pooled connections and array types. The JDBCOP is meant for use with the Foundation Profile or its supersets.

Revision Exercises

1. Name three main Optional API supported by J2ME.
2. Using a simple code show how MMAPI is initialized.
3. State the main functions of Wireless Messaging API

11. Mobile video,MMAPI and WMA.

Objectives:

At the end of this topic the students should be able to:

- Understand Mobile video and MMAPI.
- Understand the supported protocols in MMAPI.
- Create a simple J2ME application that uses MMAPI supported by J2ME.
- Understand Wireless Messaging API in J2ME
- Create simple SMS applications.

Introduction

Using MMAPI, you can easily develop robust and useful Java mobile video applications. Multimedia processing using MMAPI can be considered in two parts:

- Data delivery protocol handling
- Data content handling

Data delivery protocol handling involves reading data from a source (from a capture device or a file, for instance) into a media-processing system. *Data content handling* involves processing the media data (that is, parsing or decoding it) and rendering the media to an output device -- to a video display, for example.

MMAPI provides two high-level classes to help you with these tasks. The javax.microedition.media.protocol.DataSource class deals with data delivery protocol handling, and the javax.microedition.media.Player interface deals with data content handling. To help the developer, a Manager class

(javax.microedition.media.Manager) creates Players from DataSources, locators, and InputStreams

MIDP 2.0 , along with the optional Mobile Media API 1.1 (MMAPI), offers a range of multimedia capabilities for mobile devices, including playback and recording of audio and video data from a variety of sources. Of course, not all mobile devices support all the options, and MMAPI is designed in such a way that it takes full advantage of the capabilities that are available, while ignoring those that it cannot support. MIDP 2.0 comes with a subset of the MMAPI which ensures that if a device does not support MMAPI, you can still use a scaled down version. This scaled down version only supports audio (including tones) and excludes anything to do with video or images.

Mobile Media API (MMAPI) background

MMAPI defines the superset of the multimedia capabilities that are present in MIDP 2.0. It started life as JSR 135 and is currently at version 1.1.

The MMAPI is built on a high-level abstraction of all the multimedia devices that are possible in a resource-limited device. This abstraction is manifest in three classes that form the bulk of operations that you do with this API.

These classes are the Player and Control interfaces, and the Manager class. Another class, the DataSource abstract class, is used to locate resources, but unless you define a new way of reading data you will probably never need to use it directly.

We use the Manager class to create Player instances for different media by specifying **DataSource** instances. The Player instances thus created are configurable by using Control instances.

For example, almost all Player instances would theoretically support a **VolumeControl** to control the volume of the Player.

Figure 10-1 shows this process.

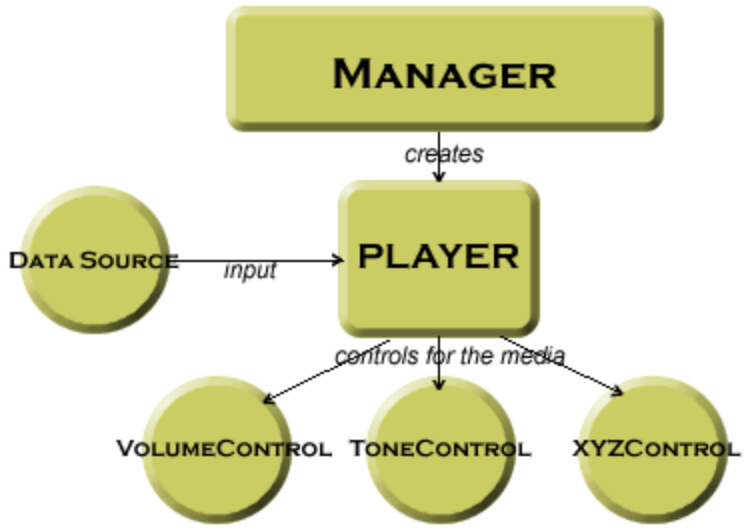


Figure 10-1: Player creation and management

Manager is the central class for creating players and it provides three methods to indicate the source of media. These methods, all static, are **createPlayer(DataSource source),**
createPlayer(InputStream stream, String type)

and createPlayer(String locator). The last method is interesting because it provides a URI style syntax for locating media. For example, if you wanted to create a Player instance on a web based audio file, you can use createPlayer("http://www.yourwebsite.com/audio/song.wav").

Similarly, to create a media Player to capture audio, you can use createPlayer("capture://audio"); and so on. Table 10.1 shows the supported syntax with examples.

Media Type	**Example syntax**
Capture audio	"capture://audio" to capture audio on the default audio capture device or "capture://devmic0?encoding=pcm" to capture audio on the devmic0 device in the PCM encoding
Capture video	"capture://video" to capture video from the default video capture device or "capture://devcam0?encoding=rgb888&width=100&height=50"

	to capture from a secondary camera, in rgb888 encoding mode and with a specified width and height
Start listening in on the built-in radio	"capture://radio?f=105.1&st=stereo" to tune into 105.1 FM frequency and stereo mode
Start streaming video/audio/text from an external source	"rtp://host:port/type" where type is one of audio, video or text
Play tones and MIDI	"device://tone" will give you a player that you can use to play tones or "device://midi" will give you a player that you can use to play MIDI

Table 11. List of supported protocols and example syntax

A list of supported protocols for a given content type can be retrieved by calling the method **getSupportedProtocols(String contentType)** which returns a String array. For example, if you call this method with the argument "audio/x-wav" it will return an array with three values in it: http, file and capture for the wireless toolkit.

This lets you know that you can retrieve the content type "audio/x-wav",by using http and file protocols, and capture it using the capture protocol.

Similarly, a list of supported content types for a given protocol can be accessed by calling the method **getSupportedContentTypes(String protocol).**

Thus, calling getSupportedContentTypes("capture") will return audio/x-wav and video/vnd.sun.rgb565 for the wireless toolkit, indicating that you can capture standard audio and

rgb565 encoded video. Note that passing null in any of these methods will return all supported protocols and content types respectively.

Once a Player instance is created using the Manager class methods, it needs to go through various stages before it can be used.

Upon creation, the player is in an UNREALIZED state and must be REALIZED and PREFETCHED before it can be STARTED. Realization is the process in which the player examines the source or destination media resources and has enough information to start acquiring them. Prefetching happens after realization and the player actually acquires these media resources. Both realization and prefetching processes may be time- and resource-consuming, but doing them before the player is started ensures that there is no latency when the actual start happens. Once a player is started, using the start() method, and is processing media data, it may enter the PREFETCHED state again when the media processing stops on its own (because the end of the media was reached, for example), you explicitly call the stop() method on the Player instance, or when a predefined time (called TimeBase) is reached. Going from STARTED to PREFETCHED state is like pausing the player, and calling start() on the Player instance restarts from the previous paused point.

The start() method implicitly calls the prefetch() method (if the player is not in a PREFETCHED state), which in turn calls the realize() method (if the player is not in a REALIZED state). A player can go into the CLOSED state if you call the close() method on it, after which the Player instance cannot be reused. Instead of closing, you can deallocate a player by calling deallocate(), which returns the player to the REALIZED state, thereby releasing all the resources that it would have acquired.

Figure 2 shows the various states and the transitions between them.

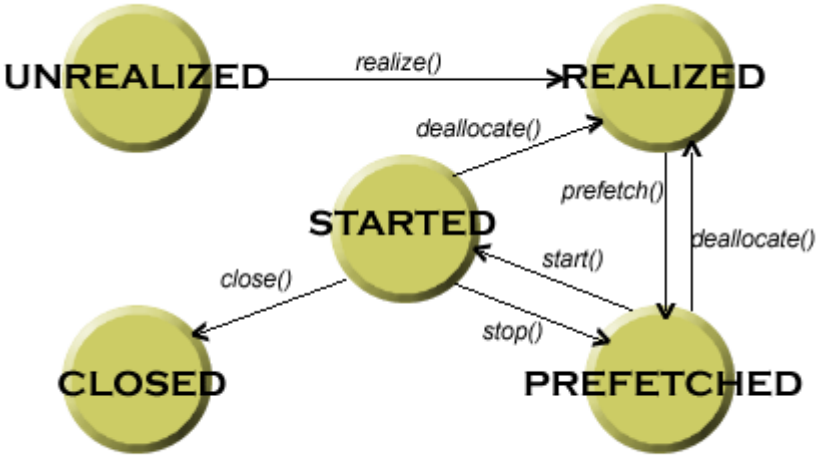


Figure 2. Media player states and their transitions.

Notification of the transitions between different states can be delivered to attached listeners on a player. A Player instance allows you to attach a PlayerListener by using the method

addPlayerListener(PlayerListener listener)

Almost all transitions states are notified to the listener via the method
playerUpdate(Player player, String event, Object eventData).

A player also enables control over the properties of the media that it is playing by using *controls*. A control is a media processing function that may be typical for a particular media type.

For example, a VideoControl controls the display of video, while a MIDIControl provides access to MIDI devices' properties. There are, of course, several controls that may be common across different media, VolumeControl being an example.

MIDP 2.0 contains a subset of the broad MMAPI 1.1.

This is to ensure that devices that only support MIDP 2.0 can still use a consistent method of discovery and usage that can scale if the broader API is present. The subset only supports tones and audio with only two controls for each,

ToneControl and **VolumeControl.** Additionally, datasources are not supported, and hence, the Manager class in MIDP 2.0 is simplified and does not provide the createPlayer(DataSource source) method.

Using Mobile Media API (MMAPI)

All multimedia operations, whether simple audio playback or complex video capture, will follow similar patterns.The Manager class will be used to create a Player instance using a String locator.

The Player will then be realized, prefetched and played till it is time to close it. There are small differences.

Figure 3 shows part of the operation of this simple audio file playback.

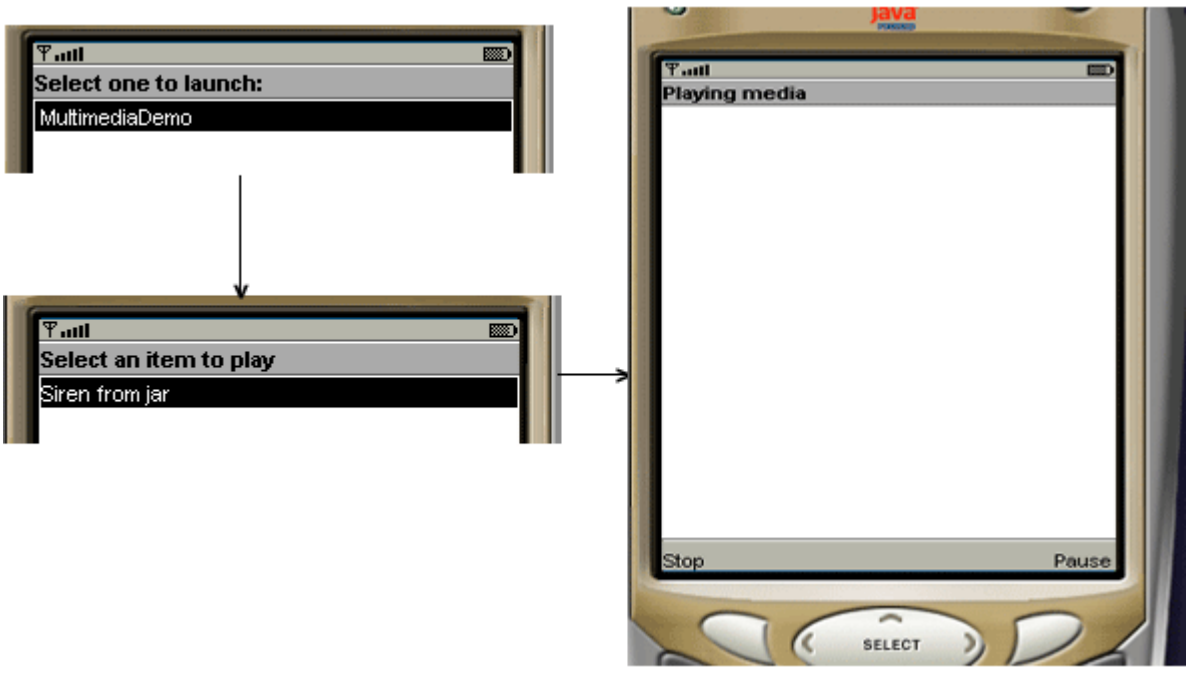


Figure 3. Simple audio file playback

When the user launches the MIDlet, he is given the option of playing the only item in the list, which is a "Siren from jar" item. On selecting this item, the screen changes to show the text "Playing media" and two commands become available to the user: pause and stop. The media starts playing in the background and the user can pause the audio or stop and return to the one item list.

The following code can be used.

```
package com.j2me.TheMedia;
import java.util.Hashtable;
import java.util.Enumeration;
import javax.microedition.lcdui.Item;
import javax.microedition.lcdui.List;
import javax.microedition.lcdui.Form;
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.media.Player;
import javax.microedition.media.Control;
import javax.microedition.media.Manager;
import javax.microedition.media.PlayerListener;
public class MediaMIDlet extends MIDlet  implements CommandListener, PlayerListener {
private Display display;
private List itemList;
private Form form;
private Command stopCommand;
private Command pauseCommand;
private Command startCommand;
```

```
private Hashtable items;
private Hashtable itemsInfo;
private Player player;
public MediaMIDlet() {
    display = Display.getDisplay(this);
    // creates an item list to let you select multimedia files to play
    itemList = new List("Select an item to play", List.IMPLICIT);

    // stop, pause and restart commands
    stopCommand = new Command("Stop", Command.STOP, 1);
    pauseCommand = new Command("Pause", Command.ITEM, 1);
    startCommand = new Command("Start", Command.ITEM, 1);

    // a form to display when items are being played
    form = new Form("Playing media");

    // the form acts as the interface to stop and pause the media
    form.addCommand(stopCommand);
    form.addCommand(pauseCommand);
    form.setCommandListener(this);

    // create a hashtable of items
    items = new Hashtable();

    // and a hashtable to hold information about them
    itemsInfo = new Hashtable();

    // and populate both of them
    items.put("Siren from jar", "file://siren.wav");
    itemsInfo.put("Siren from jar", "audio/x-wav");
  }
```

```
public void startApp() {
  // when MIDlet is started, use the item list to display elements
  for(Enumeration en = items.keys(); en.hasMoreElements();) {
  itemList.append((String)en.nextElement(), null);
}
  itemList.setCommandListener(this);
  // show the list when MIDlet is started
  display.setCurrent(itemList);
}
public void pauseApp() {
  // pause the player
  try {
    if(player != null) player.stop();
  } catch(Exception e) {}
}

public void destroyApp(boolean unconditional) {
  if(player != null) player.close(); // close the player
}
public void commandAction(Command command, Displayable disp) {
  // generic command handler
  // if list is displayed, the user wants to play the item
  if(disp instanceof List) {
    List list = ((List)disp);
    String key = list.getString(list.getSelectedIndex());
    // try and play the selected file
    try {
      playMedia((String)items.get(key), key);
    } catch (Exception e) {
      System.err.println("Unable to play: " + e);
      e.printStackTrace();
    }
```

```
  } else if(disp instanceof Form) {
    // if showing form, means the media is being played
    // and the user is trying to stop or pause the player
    try{
      if(command == stopCommand) { // if stopping the media play
        player.close(); // close the player
        display.setCurrent(itemList); // redisplay the list of media
        form.removeCommand(startCommand); // remove the start command
        form.addCommand(pauseCommand); // add the pause command
      } else if(command == pauseCommand) { // if pausing
        player.stop(); // pauses the media, note that it is called stop
        form.removeCommand(pauseCommand); // remove the pause command
        form.addCommand(startCommand); // add the start (restart) command
      } else if(command == startCommand) { // if restarting
        player.start(); // starts from where the last pause was called
        form.removeCommand(startCommand);
       form.addCommand(pauseCommand);
      }
    } catch(Exception e) {
      System.err.println(e);
    }
  }
}
/* Creates Player and plays media for the first time */
private void playMedia(String locator, String key) throws Exception {
  // locate the actual file, we are only dealing
  // with file based media here
  String file = locator.substring(
    locator.indexOf("file://") + 6,
    locator.length());
  // create the player
  // loading it as a resource and using information about it
```

```
    // from the itemsInfo hashtable
    player = Manager.createPlayer(
    getClass().getResourceAsStream(file), (String)itemsInfo.get(key));
    // a listener to handle player events like starting, closing etc
    player.addPlayerListener(this);
    player.setLoopCount(-1); // play indefinitely
    player.prefetch(); // prefetch
    player.realize(); // realize
    player.start(); // and start
  }
  /* Handle player events */
  public void playerUpdate(Player player, String event, Object eventData) {
    // if the event is that the player has started, show the form
    // but only if the event data indicates that the event relates to newly
    // stated player, as the STARTED event is fired even if a player is
    // restarted. Note that eventData indicates the time at which the start
    // event is fired.
    if(event.equals(PlayerListener.STARTED) &&
      new Long(0L).equals((Long)eventData)) {
      display.setCurrent(form);
    } else if(event.equals(PlayerListener.CLOSED)) {
      form.deleteAll(); // clears the form of any previous controls
    }
  }
}
```

You can now add to add playback for other media. To start, the MIDlet displays a list of items that can be played. At the moment, it only contains a single item called "Siren from jar". Notice that in the code, "Siren from jar" corresponds to a file-based access. This implies that the actual location of this media will be in the MIDlet jar file. When the user selects this item, a Player object is created specifically for it in the playMedia() method.

This method loads this player, attaches a listener to it, prefetches it, realizes it and finally, starts it.

Because the listener for the Player is the MIDlet class itself, the playerUpdate() method catches the player events. Thus, when the user starts hearing the siren, the Form is displayed, allowing the user to stop or pause it. Stop takes the user back to the list, while pause pauses the siren and replays from the paused marker when restarted.

Having created this generic class, it is now fairly easy to add other types of media to it. Besides audio, video is the primary media that would be played. To allow the MediaMIDlet to play video, the only change that needs to be made is in the playerUpdate() method, to create a video screen. This is shown in the following code snippet, with the changes highlighted in bold.

```
/* Handle player events */
  public void playerUpdate(Player player, String event, Object eventData) {

    // if the event is that the player has started, show the form
            // but only if the event data indicates that the event relates to newly
            // stated player, as the STARTED event is fired even if a player is
            // restarted. Note that eventData indicates the time at which the start
            // event is fired.
            If(event.equals(PlayerListener.STARTED) &&
              new Long(0L)Equals((Long)eventData)) {

      // see if we can show a video control, depending on whether the media
                        // is a video or not
                        VideoControl vc = null;
                        if((vc = (VideoControl)player.getControl("VideoControl")) != null) {
                          Item videoDisp =
                                  (Item)vc.initDisplayMode(vc.USE_GUI_PRIMITIVE, null);
                          form.append(videoDisp);
```

```
                }

                display.setCurrent(form);
        } else if(event.equals(PlayerListener.CLOSED)) {

          form.deleteAll(); // clears the form of any previous controls
        }
 }
```

The change allows you to play video files with the help of this MediaMIDlet as well. If the method determines that the player has a VideoControl, it exposes it by creating a GUI for it. This GUI is then attached to the current form.

To see the list of video files supported by a device, use the

Manager.getSupportedContentTypes(null) method.

Wireless Messaging API in J2ME

The Wireless Messaging API, or WMA, is an optional API that enables MIDP applications to leverage the utility of SMS. In addition, the API can be used for receiving and processing Cell Broadcast Service (CBS) messages. WMA 2.0, recently finalized, adds the support for Multimedia Message Service (MMS) messages. It becomes easy for MIDP applications to send and receive images, video, and other multimedia content.

Short Message Service, or SMS, is one of the most widely available and popular services for cell phone users. Often sold as text messaging service, it is the ability to send short messages between phone users.

The ubiquitous nature of SMS means that it is supported worldwide on cellular networks of (almost) any technology, including but not limited to TDMA, CDMA, WCDMA, GSM, GPRS, CDMA2000-1X, CDMA2000-MX, and EDGE.

Messages can now be sent and received between phones without

- Having Internet access from the phone (potentially additional cost service)
- Going through an intermediary server (potentially from the carrier at additional cost)

- Being restricted in terms of routing by the carrier's network.

SMS messages are sent through a store-and-forward network, and messages will queue up until the receiving device is available. This can be used to great advantage for some applications. Through SMTP gateway services, it is even possible to send SMS messages using e-mail and receive SMS messages as e-mail.

The possible applications are unlimited. Chat-type applications are the obvious for SMS, but interactive gaming, event reminders, e-mail notification, mobile access to corporate resources, and informational service are a sample of other opportunities.

WMA makes SMS available to MIDP developers. In other words, WMA provides a generalized direct point-to-point communications mechanism for MIDP applications.

WMA and SMS

When a message is sent and received via WMA, there are two major benefits. First, the message can now be significantly larger in size than a single SMS message. Second, the content is no longer restricted to simple text messages. Silently, the WMA API will do the following:

- Encode a binary message and transmit it through SMS.
- Cut up a long message into segments, and send it via multiple (up to three) SMS messages.

Two current and finalized JSRs are relevant to WMA, and these are

- 120 Wireless Messaging API http://jcp.org/jsr/detail/120.jsp
- 205 Wireless Messaging API 2.0 http://jcp.org/jsr/detail/205.jsp

You can also create a server mode connection. Instead, if you were to open the connection using the following:

MessageConnector msgConn = (MessageConnector) Connector.open("sms:// :1234");

you would end up with a server mode connection.

Once you have a MessageConnection, you can use it to

- Create messages.
- Receive messages (server mode only).
- Send messages.
- Obtain segmentation information on a message.

Creating New Messages

A MessageConnection is the class factory for messages. There is no way to create a message other than using a MessageConnection. which actually has two variations:

Message newMessage(String messageType);

Message newMessage(String messageType, String address);

Sending Text SMS Messages

Sending a text message is similar to sending binary messages. First, create an empty message using MessageConnection's **newMessage()** method, specifying MessageConnection.TEXT_MESSAGE as its type.

Next, set the payload with the text string that you want to send:

public void setPayloadText(String data);

Finally, use the send() method of MessageConnection to send the message. Here is a code fragment that you may use to open a connection and send an SMS text message:

```
MessageConnection conn =
(MessageConnection) Connector.open("sms://5550001:1234");
TextMessage txtmessage = (TextMessage) conn.newMessage(
MessageConnection.TEXT_MESSAGE);
txtmessage.setPayloadText(msgString);
conn.send(txtmessage);
```

You can also send large text messages that span multiple SMS messages. The WMA API will handle segmentation at the sending end and reassembly at the receiving end.

Revision Exercises

1. Differentiate between Mobile video and MMAPI.
2. Discuss different protocols supported in MMAPI.
3. Create a simple J2ME application that uses MMAPI supported by J2ME.
4. Discuss the Wireless Messaging API in J2ME
5. Create simple SMS applications.