INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)

Dundigal, Hyderabad - 500 043

INFORMATION TECHNOLOGY

MOBILE APPLICATION DEVELOPMENT

Prepared By:
D.Rahul
Assistant Professor

UNIT-I

- **J2ME Overview:** Java 2 Micro Edition and the world of Java, Inside J2ME, J2ME and Wireless Devices small Computing Technology. Wireless Technology, Radio Data Networks, Microwave Technology, Mobile Radio Networks, Messaging, Personal Digital Assistants

Overview

- Introduction of Mobile Technology
- What is J2ME
- Java 2Micro edition and the world of Java
- Inside J2ME (How J2ME is organized)
- J2ME profiles & Wireless Devices
- What J2ME Isn’t & other Java Platforms
- Wireless Technology

Introduction of Mobile Technology

- The goals Mobile Technology
  - Connecting people
  - Information sharing
  - Internet access
  - Entertainment

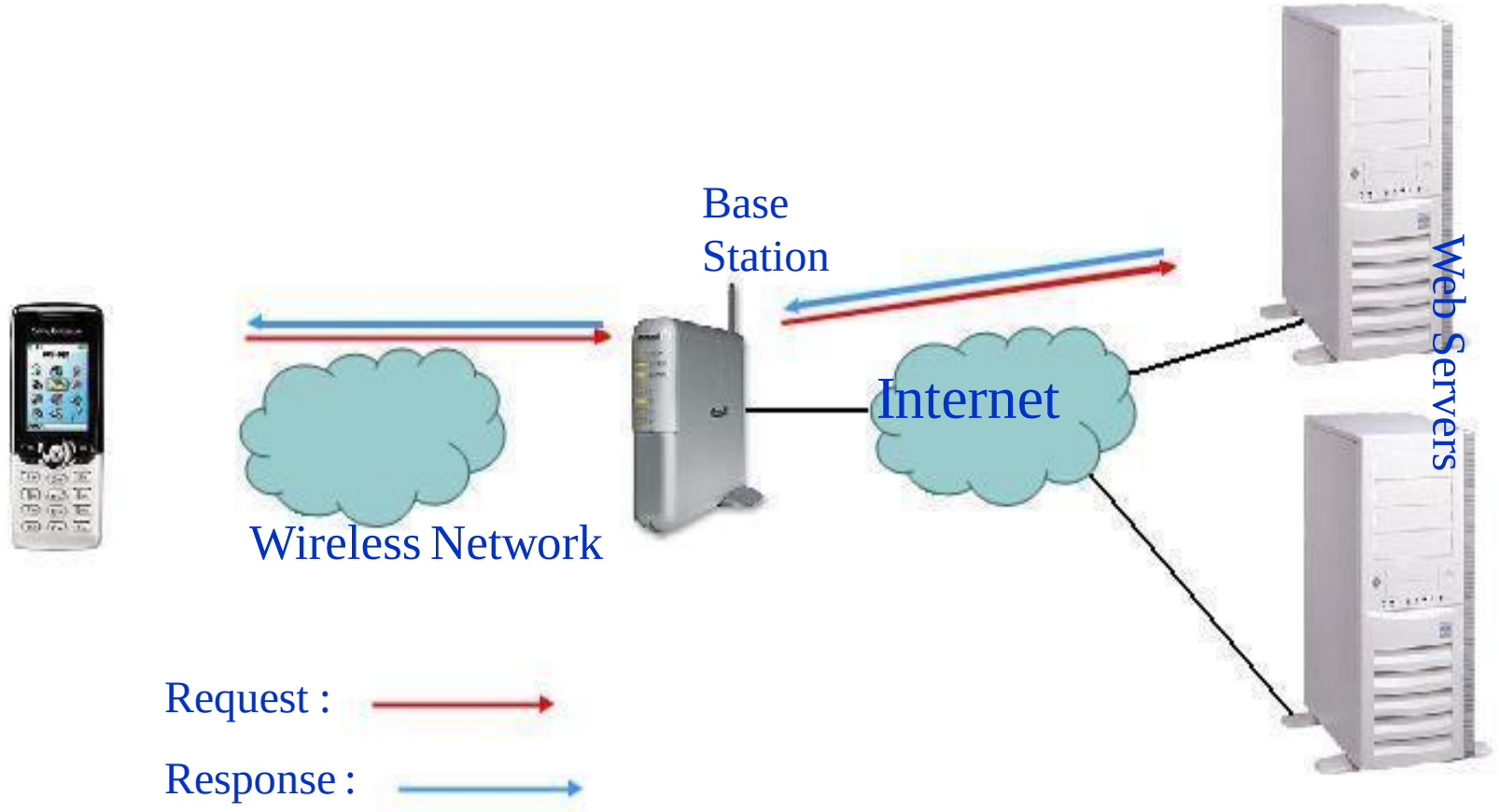
  with the most importance words – “at any time, any where”

Introduction of Mobile

- Includes
  - Notebook
  - Palmtops
  - PDAs
  - Mobile Phones
  - Tablet PCs
  - And more ...

Introduction of Mobile

- The Internet Scenario for retrieving information in a wireless network



What is J2ME

J2ME is a family of specifications that defines various downsized versions of the standard Java 2 platform; these downsized versions can be used to program consumer electronic devices ranging from cell phones to highly capable Personal Data Assistants (PDAs), smart phones, and set-top boxes.

Java 2Micro edition and the world of Java

Java programming language developed by Sun Microsystems

- Required a Virtual machine to interpret the source codes and generate bytecode
- Syntax is similar to C++
- Platform independent feature

Java 2Micro edition

- Java includes three different editions
  - J2SE (Java 2 Standard Edition)
  - J2EE (Java 2 Enterprise Edition)
  - J2ME (Java 2 Micro Edition)
- The above three editions target for different devices or systems

Java 2Micro edition

- J2SE (Desktop-based applications)
  - Provides a complete environment for applications development on desktops and servers
  - The foundation of J2EE
  - J2SE 1.5 (Tiger) is available now!

Java 2Micro edition

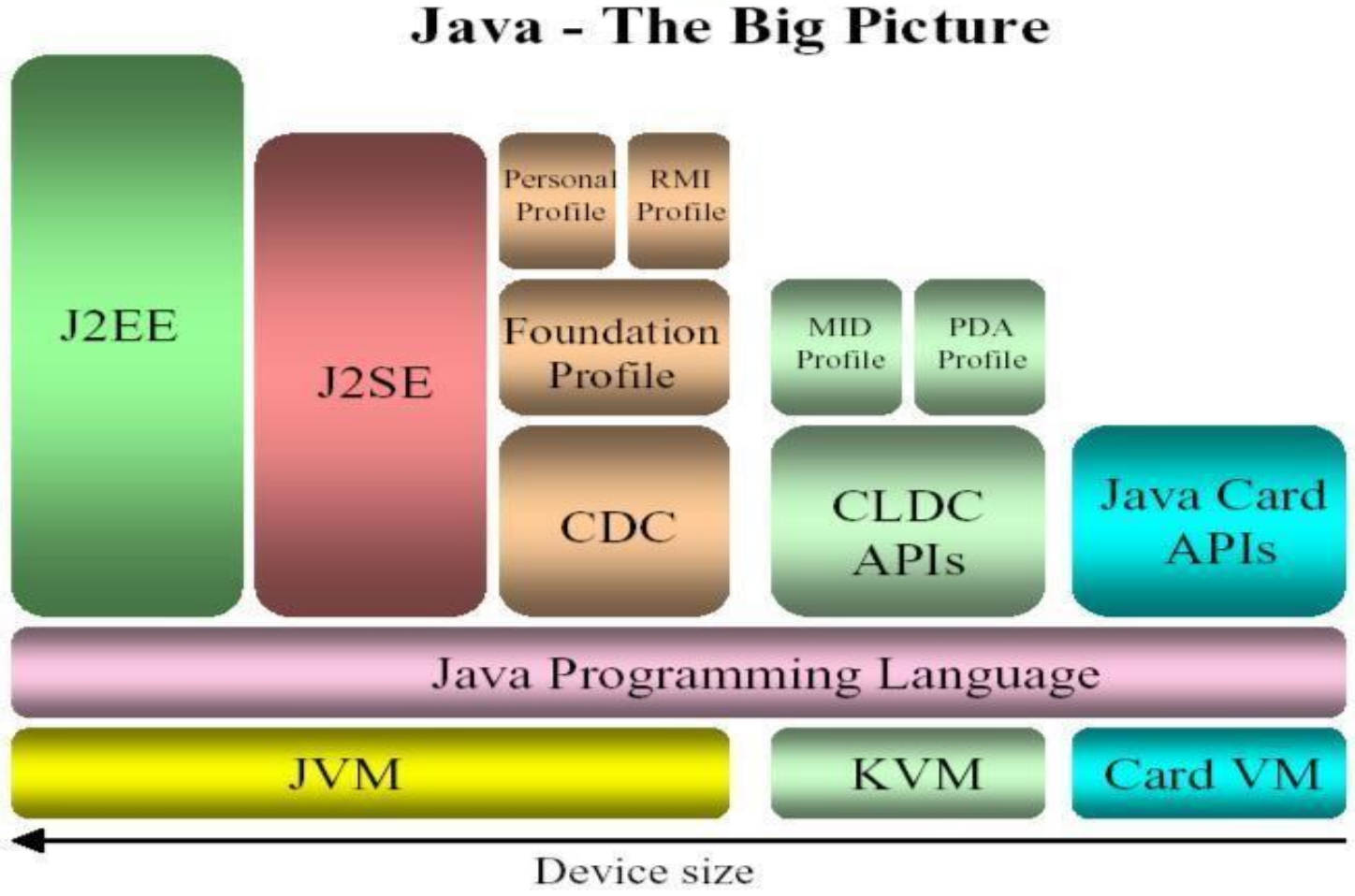
- J2EE (Server-based applications)
  - Target for business use
  - Large scale of systems which may contain tens of servers and millions of users
  - Web based services
  - Machines are high performance

Java 2Micro edition

- J2ME (For handheld and embedded devices)

  - The Micro Edition of the Java 2 Platform provides an application environment that specifically addresses the needs of commodities in the vast and rapidly growing consumer and embedded space, including mobile phones, pagers, PDAs, set-top boxes, and vehicle telematics systems

J2ME and the world of Java

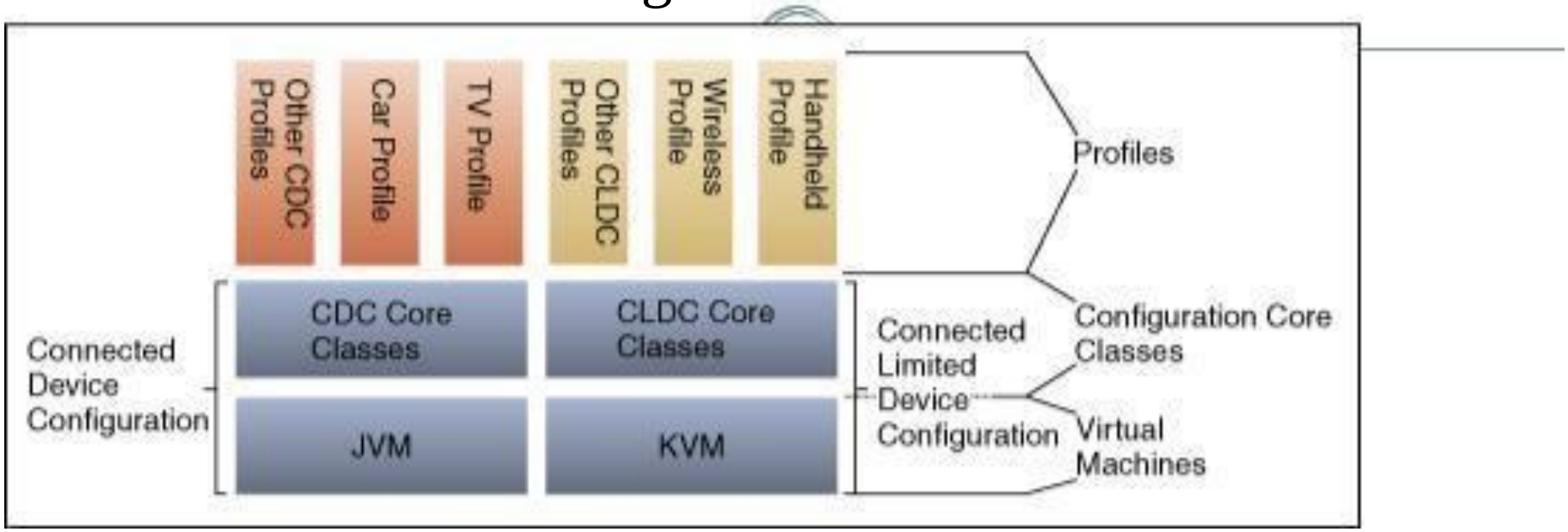


How J2ME is Organized

J2ME is organized by two categories

- Configuration :
  - is a complete Java runtime environment:
    - Java virtual machine (VM) to execute Java.
    - Set of core Java runtime classes
    - Interface to the underlying system
  - J2ME supports Two basic configurations CDC- Connected Device Configuration
    CLDC – Connected Limited Device Configuration

How J2ME is Organized



Principle: Different hardware corresponds different JVMs, hardware which base on CDC have powerful JVMs, and hardware which base on CIDC have KVM to support.

Connected Limited Device Configuration (CLDC)

CLDC is aimed at the low end of the consumer electronics range.

CDLC (***Connected Limited Device Configuration):***160 - 512 KB of total memory

16-bit or 32-bit processor
Low power consumption and often operating with battery power
Connectivity with limited bandwidth

Connected Device Configuration (CDC)

- CDC addresses the needs of devices that lie between those addressed by CLDC and the full desktop systems running J2SE.
  - 2 MB or more memory for Java platform.
  - 32-bit processor.
  - High bandwidth network connection.
  - full-featured Java 2 virtual machine (CVM).
  - 17 packages.
  - Use for devices like Palms.

J2ME Profiles

- ***Mobile Information Device Profile*** **(MIDP)** - CLDC-based, used for running applications on cell phones and interactive pagers with small screens, wireless HTTP connectivity, and limited memory.
- ***Personal Digital Assistant Profile*** **(PDAP)** – CLDC-based, extends MIDP with additional classes and features for more powerful handheld devices.
- ***Foundation Profile*** **(FP)** – CDC-based, extends the CDC with additional J2SE classes.
- ***Personal Basis Profile*** **(PBP)** - extends the FP with lightweight (AWT-derived) user interface classes and a new application model.
- ***Personal Profile*** extends the PBP with applet support and heavyweight UI classes.

- The CLDC-profile used today:

  MIDP (Mobile Information Device Profile)

- The MIDP defines a platform for dynamically and securely deploying optimized, graphical, networked applications.

- The MIDP specification was defined through the Java Community Process (JCP) by players like: Motorola, Nokia, Ericsson, Research in Motion, and Symbian.

MIDP – MID Profile

- MIDP is targeted at a class of devices known as *mobile information devices* (MIDs).

- Minimal characteristics of MIDs:
  - Enough memory to run MIDP applications
  - Display of at least 96 X 56 pixels, either monochrome or color
  - A keypad, keyboard, or touch screen
  - Two-way wireless networking capability

J2ME and Wireless Devices

- WAP (Wireless Application Protocol) forum is the initial industry group that set out to create standard s for wireless technology
- Initially Ericsson, Motorola, nokia and Unwired Planet formed the WAP forum in 1997 and it has grown to include nearly all mobile device manufacturers mobile network providers and developers
- WAP standard is an enhancement of HTML XML and TCP/IP., it includes WML specification(wireless Markup language) and also WTAI (Wireless Telephony Application Interface)
- Many sophisticated applications designed for mobile communications devices require the device to process information beyond the capabilities of the WAP specification
- J2ME provides the standard to fill this gap

What J2ME Isn't

- Although J2ME is J2SE without some classes developers shouldn't assume that existing java applications would run in the J2ME environment because of resource constraints imposed by small computing devices

✓ The write once and run anywhere is overstated with J2ME

✓ Some J2SE applications require classes which are not available in J2ME

✓ Small computing devices use JVM or KVM based on configuration of the devices

✓ MIDlets are controlled by the AMS(Application Management software) not like J2SE

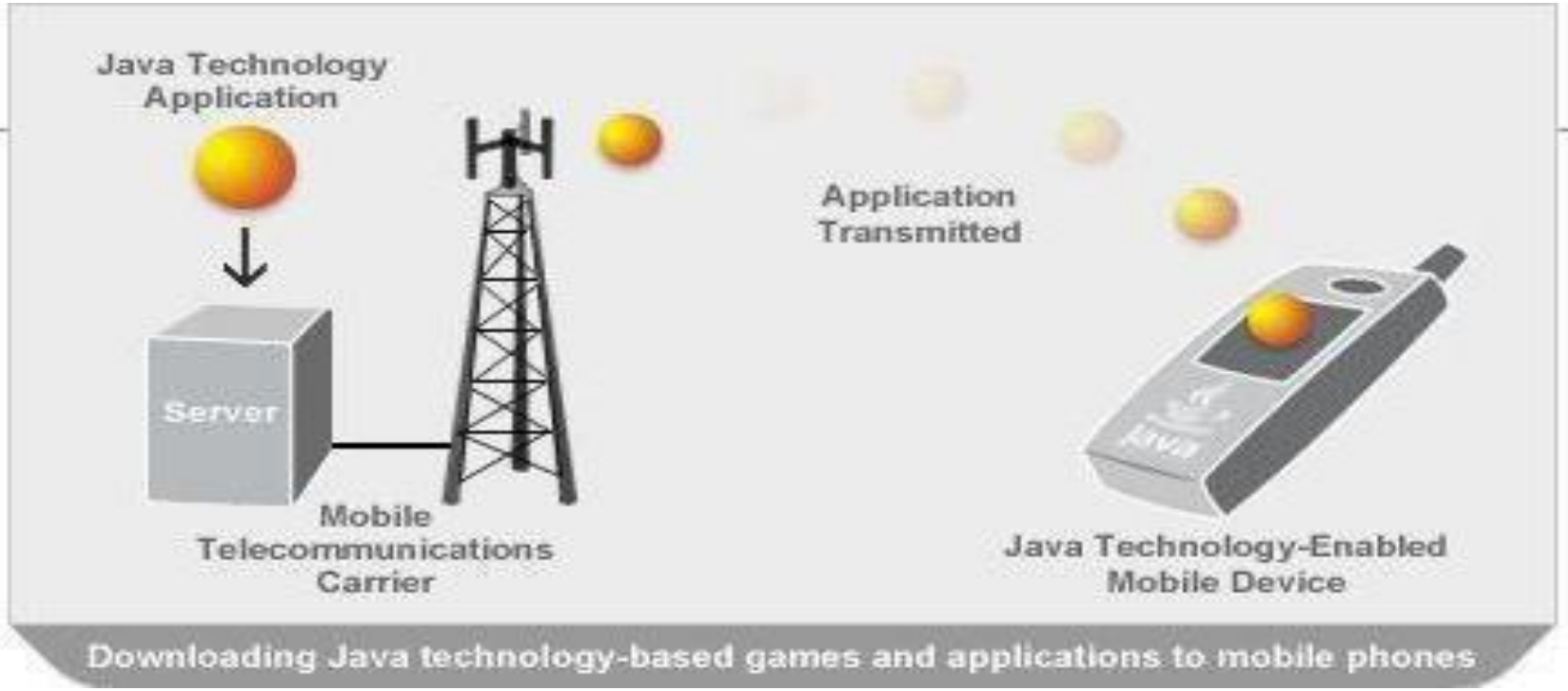
Other Java Plat

- Embedded Java is the platform used for small computing devices dedicated to one purpose and have a 32 bit processor and 512kb ROM and RAM

- Java Card is the platform used for smart cards

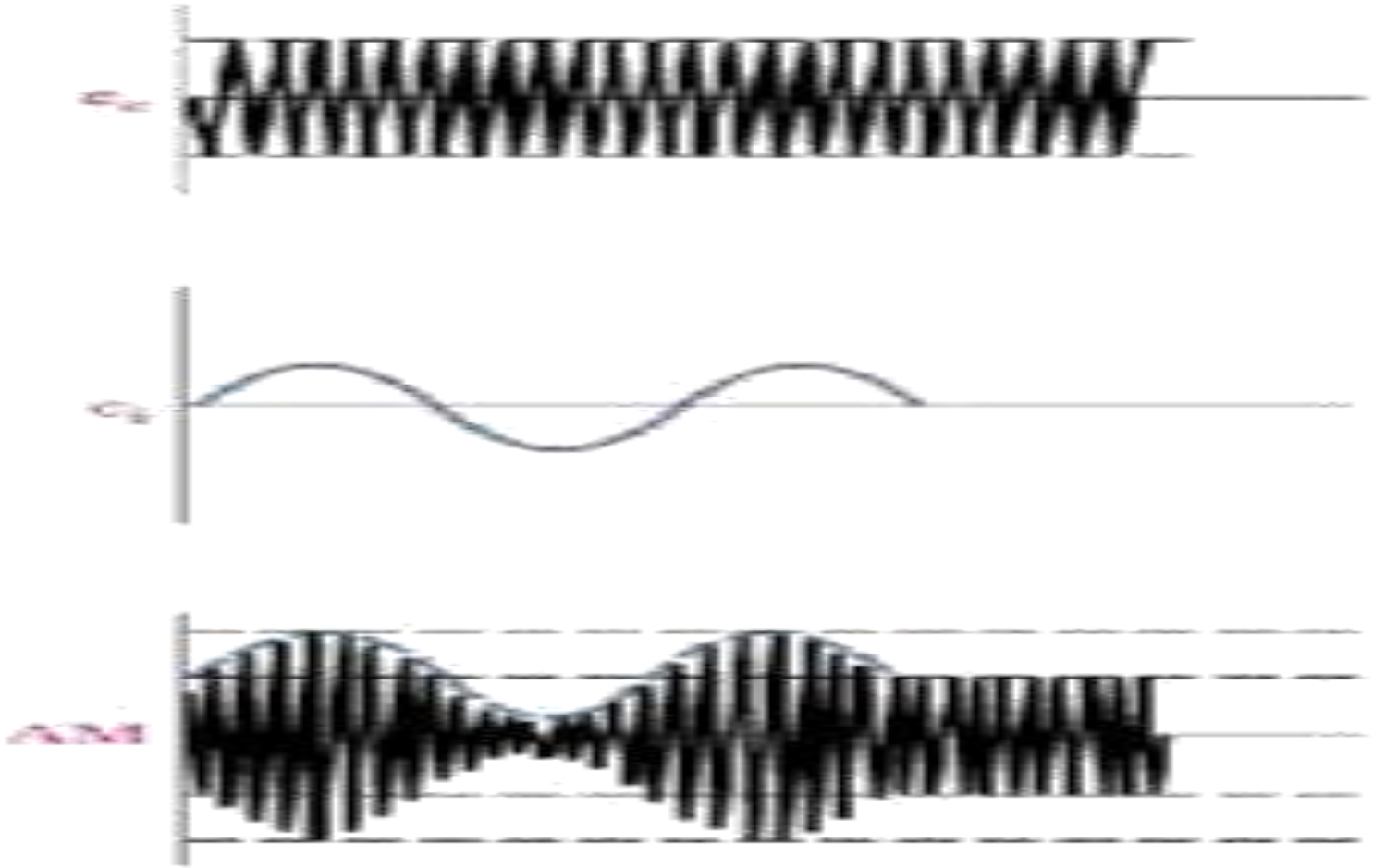
- Personal Java is the platform used for small computing devices that have a maximum of 2MB ROM and 1 MB RAM

How does J2ME work?

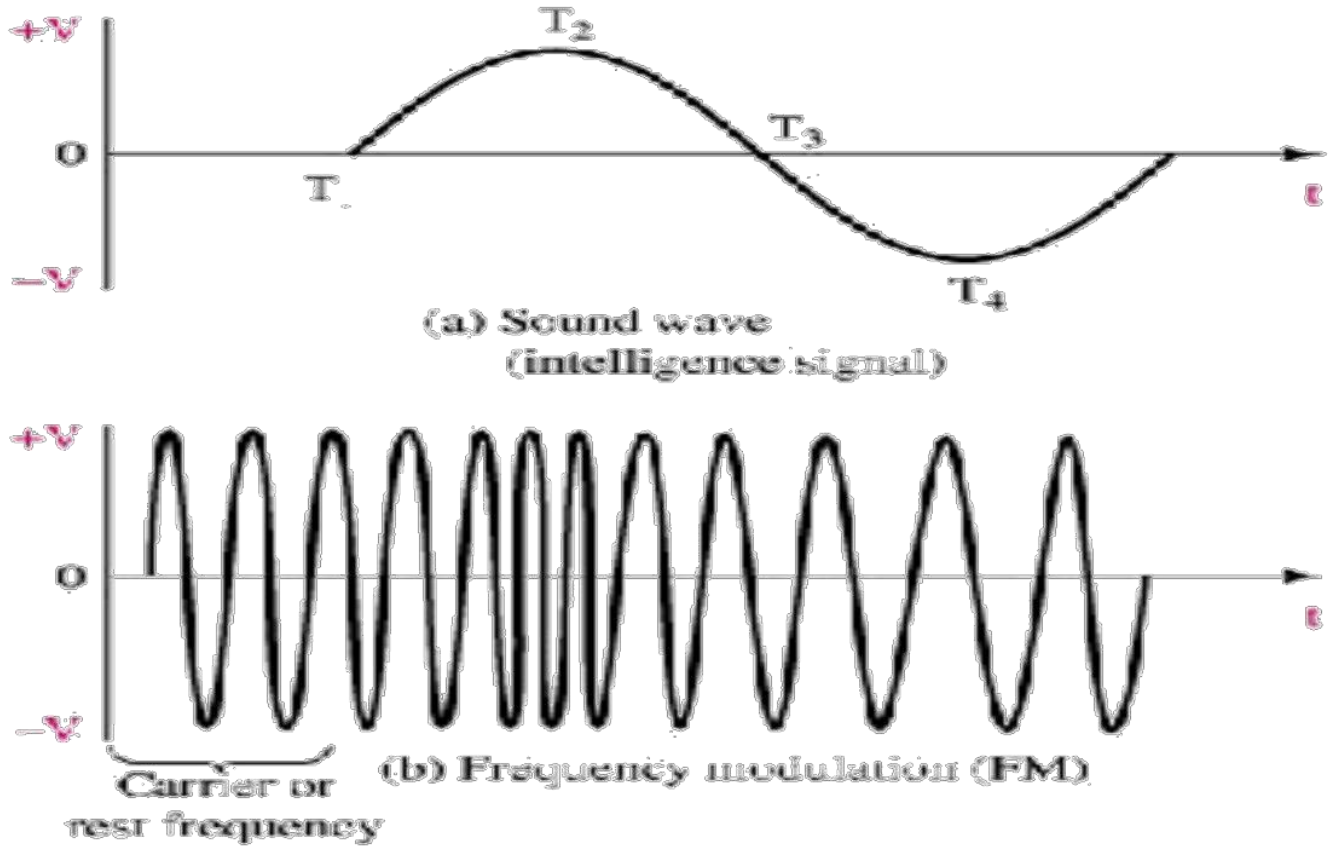


The latest Java-enabled mobile devices, you can view a list of applications, games, and services and choose which one interests you. The application is then sent over the air to your handset, where it is installed and instantly available to use.

AM Waveforms



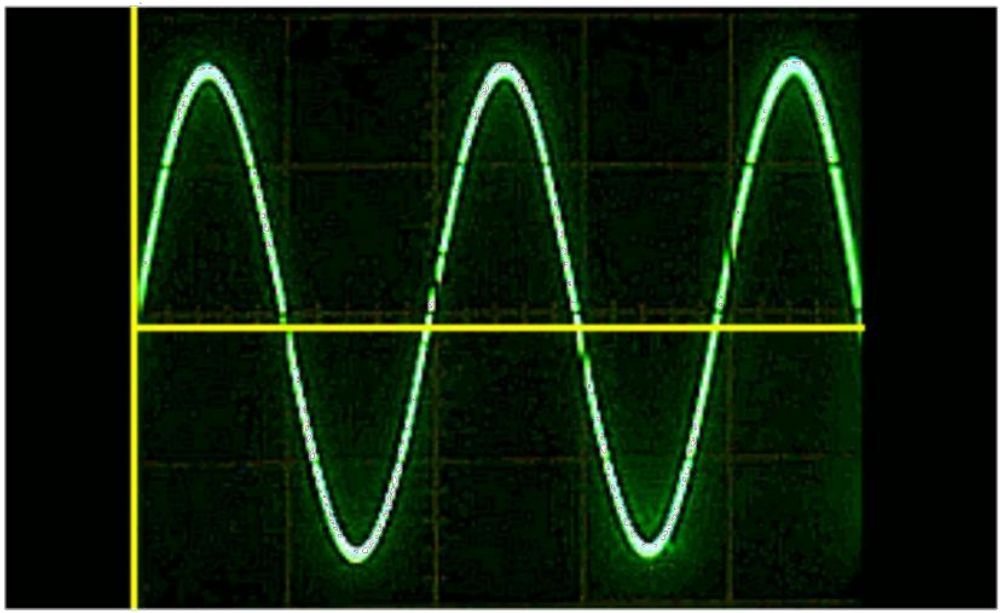
FM Waveforms



The carrier....

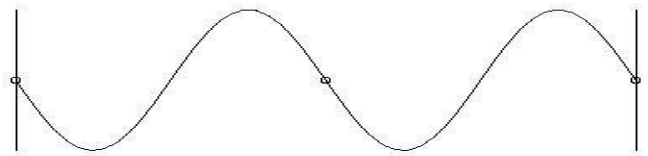
- A high frequency ‘carrier’ takes information from one place to another.
- The ‘carrier’ is considered to be a ‘radio frequency’ or RF.
- The ‘information’ is attached to the carrier using AM, FM, or PCM or another method.

The carrier is a Sine Wave

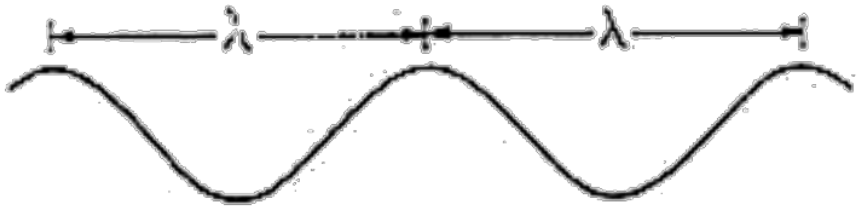


Carrier velocity = v

- The carrier travels at the speed of light
- 186,000 miles/second
- 300,000,000 meters/second



Wavelength = v/f



F = 1290 KHz (WNBF Radio)

$$\lambda = v/f = 300{,}000{,}000/1{,}290{,}000$$

$$= 232.5 \text{ meters}$$

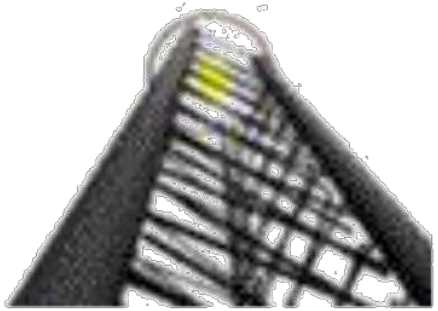
Amplitude Modulation

- Includes Broadcast Radio 540 KHz to 1640KHz
- Citizen Band
- Amateur Radio
- Television Video

Frequency Modulation

- Broadcast FM from 88 MHz to 108 MHz
- Aircraft, Marine, Taxi
- Some digital transmissions
- Television Audio
- All Satellite Television

Electro Magnetic Radiation



Powerful Transmitters..

- 50 Kilo Watts into a large tower antenna
- Can travel around the globe under certain weather conditions.
- The signal strength is affected by many conditions including humidity, cloud cover and content, time of day, and the terrain.

The receiver....

- Always has some type of antenna to catch the signal transmitted.
- A radio direction finder has a ‘directional’ antenna.
- The weak signal from the antenna is amplified.
- The information is removed from the carrier.

Noise is undesirable..

- Noise comes from many sources
- It can interfere with the signal
- AM is especially vulnerable
- FM is preferred due to its immunity
- TV Sound is FM, and generally quiet
- AM Radio is often noisy- especially long distance reception.

JAVA TECHNOLOGY CARRIERS

Hutchison Telecoms
Aer Wireless
AIS
KDDI
orange
Bell
LG TeleCom
SONOFON
cingular WIRELESS
中国联通 CHINA UNICOM
m1
TELECOM ITALIA
NEXTEL
中華電信
New World Mobility Better than you think
Telefonica
Sprint
NTT DoCoMo
Reliance
sonera
TELUS
DTAC makes it easy
數碼通 SHARTONE
TELIA
StarHub
T-Mobile
HK CSL
SUNDAY
vodafone
J-PHONE vodafone
WIND

Java Technology Handsets

To date, over 250 different handset models from more than 40 manufacturers have been developed with Java technology, with more than 100 million of these handsets shipped worldwide.

---http://www.java.com/en/learn/mobile_(2003)

Referenc

es

- http://jcp.org/jsr/detail/30.jsp
- http://java.sun.com/products/consumer-embedded/
- http://java.sun.com/j2me/j2me-ds.pdf
- *Topley,K. J2ME in a Nutshell -A Desktop Quick Reference*
- http://wireless.java.sun.com/
- http://www.scc-kk.co.jp/lib_scc/catalog/books/B-228/B-_228.pdf
- http://chinaunix.net/jh/26/128217.html
- http://developer.java.sun.com/developer/products/j2me/
- http://wireless.java.sun.com/midp/articles/#appmodels
- http://wireless.java.sun.com/midp/articles/getstart/
- http://www.java.com/en/learn/mobile

IMPORTANT QUESTIONS

1. Explain the differences between J2SE , J2EE & J2ME
2. Explain how J2ME balances a thin client and a thick client
3. Explain How J2ME balances local and server side processing
4. Explain J2ME Profiles
5. Explain Wireless Technology
6. Explain Cellular Digital Packet Data protocol
7. Explain Radio Data Networks & their Limitations

UNIT-II

- **J2ME Architecture and Development Environment**: J2ME Architecture, Small computing Device Requirements, Run-Time Environment, MIDlet Programming, java Language for J2ME, J2ME Software Development Kits, Hello World J2ME Style Multiple MIDLetsinaMidlet Suite, J2ME Wireless Toolkit.J2ME Best Practices and Patterns: The Reality of working in a J2ME World, Best Practices.

- The modular design of the J2ME architecture enables an application to be scaled based on constraints of a small computing device.
- J2ME architecture doesn’t replace the operating

system of a small computing device. Instead, J2ME architecture consists of layers located above the native operating system, collectively referred to as the Connected Limited Device Configuration (CLDC).environment for small computing devices.

- The J2ME architecture comprises three software layers
- **The first layer** is the configuration layer that includes the Java Virtual Machine (JVM), which directly interacts with the native operating system. The configuration layer also handles interactions between the profile and the JVM.
- **The second layer** is the profile layer, which consists of the minimum set of application programming interfaces (APIs) for the small computing device.
- **The third layer** is the Mobile Information Device Profile (MIDP).
- The MIDP layer contains Java APIs for user network connections, persistence storage, and the user interface. It also has access to CLDC libraries and MIDP libraries.

- A small computing device has two components supplied by the original equipment manufacturer (OEM).
  - These are classes and applications.
  - OEM classes are used by the MIDP to access device-specific features such as sending and receiving messages and accessing device-specific persistent data.
  - OEM applications are programs provided by the OEM, such as an address book. OEM applications can be accessed by the MIDP.
  - A word of caution: accessing OEM classes and OEM applications from the MIDP restricts the portability of a J2ME application since not all small computing device manufacturers use the same OEM classes or OEM applications.

Small Computing Device Requirements

- There are minimum resource requirements for a small computing device to run a J2ME application.
- 1. The device must have a minimum of 96 × 54 pixel display that can handle bitmapped graphics and have a way for users to input information, such as a keypad, keyboard, or touch screen.
- 2.At least 128 kilobytes (KB) of nonvolatile memory is

necessary to run Mobile Information Device (MID), and 8KB of nonvolatile memory is needed for storage of persistent application data.

- To run JVM, 32KB of volatile memory must be available. The device must also provide two-way network connectivity.

- Besides minimal hardware requirements, there are also

  minimal requirements for the native operating system.

- The native operating system must implement exception handling, process interrupts, be able to run the JVM, and provide schedule capabilities.
- Furthermore, all user input to the operating system must be forwarded to the JVM, otherwise the device cannot run a J2ME application.
- Although the native operating system doesn't need to implement a file system to run a J2ME application, it must be able to write and read persistent data (data retained when the device is powered down) to nonvolatile memory.

Run-Time Environment

- A MIDlet is a J2ME application designed to operate on an MIDP small computing device. A MIDlet is defined with at least a single class that is derived from the javax.microedition.midlet.MIDlet abstract class. Developers commonly bundle related MIDlets into a MIDlet suite, which is contained within the same package and implemented simultaneously on a small computing device. All MIDlets within a MIDlet suite are considered a group and must be installed and uninstalled as a group.

- Members of a MIDlet suite share resources of

  the host environment and share the same instances of Java classes and run within the same JVM. This means if three MIDlets from the same MIDlet suite run the same class, only one instance of the class is created at a time in the Java Virtual Machine.

- A key benefit of the relationship among MIDlet suite members is that they share the same data, including data in persistent storage such as user preferences.

- Sharing data among MIDlets exposes each MIDlet to data errors caused by concurrent read/write access to data. This risk is reduced by synchronization primitives on the MIDlet suite level that restrict access to volatile data and persistent data.
- However, if a MIDlet uses multi-threading, the MIDlet is responsible for coordinated access to the record store.

- Data cannot be shared between MIDlets that are not from the same MIDlet suite because the MIDlet suite name is used to identify data associated with the suite.

- A MIDlet from a different MIDlet suite is considered an unreliable source.

- A MIDlet suite is installed, executed, and removed by the application manager running on the device.
- The manufacturer of the small computing device provides the application manager. Once a MIDlet suite is installed, each member of the MIDlet suite is given access to classes of the JVM and CLDC by the application manager.
- Like wise access classes defined in the MIDP to interact with the user interface, network, and persistent storage.
- The application manager also makes the Java archive (JAR) file and the Java application descriptor (JAD) file available to members of the MIDlet suite.

JAR

- All the files necessary to implement a MIDlet suite must be contained within a production package called a Java archive (JAR) file.
- These files include MIDlet classes, graphic images (if required by a MIDlet), and the manifest file. The manifest file contains a list of attributes and related definitions that are used by the application manager to install the files contained in the JAR file onto the small computing device.
- Nine attributes are defined in the manifest file; all but six of these attributes are optional.

Manifest File Attribute ----- Description

- 1. MIDlet-Name- MIDlet suite name.
- 2. MIDlet-Version-MIDlet version number.
- 3.MIDlet-Vendor -Name of the vendor who supplied the MIDlet.
- 4. MIDlet-n Attribute per MIDlet.Values are MIDlet name, optional icon, and MIDlet class name.
- 5. MicroEdition-Profile Identifies the J2ME profile that is necessary to run the MIDlet.

- 6. MicroEdition-Configuration Identifies the J2ME configuration that is necessary to run the MIDlet.
- 7. MIDlet-Icon- Icon associated with MIDlet, must be in PNG image format (optional).
- 8. MIDlet-Description- Description of MIDlet (optional).
- 9. MIDlet-Info-URL- URL containing more information about the MIDlet.

Example- Manifest file List

- MIDlet-Name: Best MIDlet
- MIDlet-Version: 2.0
- MIDlet-Vendor: MyCompany
- MIDlet-1: BestMIDlet,

  /images/BestMIDlet.png, Best.BestMIDlet
- MicroEdition-Profile: MIDP-1.0
- MicroEdition-Configuration: CLDC-1.0

The Java Application Descriptor File

- include a Java application descriptor (JAD) file within the JAR file of a MIDlet suite as a way to pass parameters to a MIDlet without modifying the JAR file.
- A JAD file is also used to provide the application manager with additional content information about the JAR file to determine whether the MIDlet suite can be implemented on the device.
- A JAD file is similar to a manifest in that both contain attributes that are name:value pairs. Name:value pairs can appear in any order within the JAD file.

- There are five required system attributes for a JAD file:
- MIDlet-Name
- MIDlet-Version
- MIDlet-Vendor
- MIDlet-*n*
- MIDlet-Jar-URL

Example- JAD file list

- MIDlet-Name: Best MIDlet
- MIDlet-Version: 2.0
- MIDlet-Vendor: MyCompany
- MIDlet-Jar-URL: http://www.mycompany.com/bestmidlet.jar
- MIDlet-1: BestMIDlet,

  /images/BestMIDlet.png, Best.BestMIDlet

Attributes of JAD File

JAD File Attribute	Description
MIDlet-Name	MIDlet suite name.
MIDlet-Version	MIDlet version number.
MIDlet-Vendor	Name of the vendor who supplied the MIDlet.
MIDlet-*n*	*Attribute per MIDlet. Values are MIDlet name*, optional icon, and MIDlet class name.
MIDlet-Jar	URL Location of the JAR file.

- MIDlet-Jar-Size Size of the JAR file in bytes (optional).
- MIDlet-Data-Size Minimum size (in bytes) for persistent data storage (optional).
- MIDlet-Description Description of MIDlet (optional).
- MIDlet-Delete-Confirm Confirmation required before removing the MIDlet suite (optional).
- MIDlet-Install-Notify Send installation status to given URL (optional).

MIDlet Programming

- Programming a MIDlet is similar to creating a J2SE application in that define a class and related methods.
- Less Robust
- A MIDlet is a class that extends the MIDlet class and is the interface between application statements and the run-time environment, which is controlled by the application manager.

- A MIDlet class must contain three abstract methods that are called by the application manager to manage the life cycle of the MIDlet. These abstract methods are startApp(), pauseApp(), and destroyApp().

- The startApp() method is called by the application manager when the MIDlet is started and contains statements that are executed each time the application begins execution.
- The pauseApp() method is called before the application manager temporarily stops the MIDlet.
- The application manager restarts the MIDlet by recalling the startApp() method.
- The destroyApp() method is called prior to the termination of the MIDlet by the application manager.

Basic Shell of MIDlet

- the MIDlet class called BasicMIDletShell extends the MIDlet class. Any name can be used for a class as long as it conforms to the Java class naming convention.

- public class BasicMIDletShell extends MIDlet
- public void startApp(){
- {
- }
- public void pauseApp()
- {
- }
- public void destroyApp(boolean unconditional)
- {
- }
- }

- Both the startApp() and pauseApp() methods are public and have no return value nor parameter list.
- The destroyApp() method is also a public method without a return value.
- However, the destroyApp() method has a boolean parameter that is set to true
- if the termination of the MIDlet is unconditional, and false if the MIDlet can throw a MIDletStateChangeException telling the application manager that the MIDlet does not want to be destroyed just yet.

- At the center of every MIDlet are the MIDP API classes used by the MIDlet to interact with the user and handle data management. User interactions are managed by user interface MIDP API classes.

- These APIs enable a developer to display screens of data and prompt the user to respond with an appropriate command.
  - The command causes the MIDlet to execute one of three routines: perform a computation, make a network request, or display another screen.

- The data-handling MIDP API classes enable the developer to perform four kinds of data routines: **write and read** persistent data, **store data** in data types, **receive data from and send data** to a network, and **interact** with the small computing device's input/output features.

Event Handling

- A MIDlet is an event-based application.
- All routines executed in the MIDlet are invoked in response to an event reported to the MIDlet by the application manager.
- The initial event that occurs is when the MIDlet is started and the application manager invokes the startApp() method.

- The startApp() method in a typical MIDlet contains a statement that displays a screen of data and prompts the user to enter a selection from among one or more options. The nature and number of options is MIDlet and screen dependent.
- A Command object is used to present a user with a selection of options to choose from when a screen is displayed. Each screen must have a CommandListener.

User Interfaces

- The design of a user interface for a MIDlet

  depends on the restrictions of a small computing device.

- Some small computing devices contain resources that provide a rich user interface, while other more resource-constrained devices offer a modest user interface. A rich user interface contains the following elements, and a device

  with a minimal user interface has some subset of these elements as determined by the profile used for the device.

- A **Form** is the most commonly invoked user interface element found in a MIDlet and is used to contain other

- A **StringItem** contains text that appears on a form that cannot be changed by the user.
- A **List** is an itemized options list from which the user can choose an option.
- A **ChoiceGroup** is a related itemized options list.
- **Ticker** is text that is scrollable.

- A user enters information into a form by using the Choice element, TextBox, TextField, or DateField elements.
- The Choice element returns an option that the user selected. TextBox and TextField elements collect textual information from a user and enable the user to edit information that appears in these user interface elements.
- The DateField is similar to a TextBox and TextField except its contents are a date and time.
- An Alert is a special Form that is used to alert the user that an error has occurred.
- An Alert is usually limited to a StringItem user interface element that defines the nature of the error to the user.

Device Data

- Small computing devices don't have the resources necessary to run an onboard database management system (DBMS). In fact some of these devices lack a file system.
- Therefore, a MIDlet must read and write persistent data without the advantage of a DBMS or file system.
- A MIDlet can use an MIDP class—RecordStore—and two MIDP interfaces— RecordComparator and RecordFilter—to write and read persistent data.
- A RecordStore class contains methods used to write and read persistent data in the form of a record.
- Persistent data is read from a RecordStore by using either the RecordComparator interface or the RecordFilter interface.

Java Language for J2ME

- Stripped version
- Floating-point math is probably the most notable missing feature of J2ME.
- Floating-point math requires special processing hardware to perform floating-point calculations.
- However, most small computing devices lack such hardware and therefore are unable to process floating-point calculations. This means that your MIDlet cannot use any floating-point data types or calculations.

- The second most notable difference between the Java language used in J2SE and J2ME is the absence of support for the finalize() method. The finalize() method in J2SE is automatically called before an instance of a class terminates and typically contains statements that free previously allocated resources. However, resources in a small computing device are too scarce to process the finalize() method.

- Another dramatic difference is the reduced number of error-handling exceptions that are supported in J2ME.

- Exception handling drains system resources,

  which are precious in a small computing device and therefore the primary reason for trimming the number of error-handling exceptions.

  Typically, run-time errors are automatically

  responded to by the native operating system by restarting the small computing device.

- Changes were also made in the Java Virtual Machine that runs on a small computing device because of resource constraints. One such change occurs with the class loader.
- JVM for small computing devices requires a custom class loader that is supplied by the device manufacturer and cannot be replaced or modified.
- Another feature lacking in the JVM is the ThreadGroup class. You cannot group threads. All threads are handled at the object level, although there is a workaround . Also, you cannot call other programming languages' methods and APIs, primarily because of the memory requirements to execute such calls.
- Two other features of J2SE that are missing from J2ME are weak references and the Reflection classes.

- The standard JVM uses class file verification to protect applications from malicious code through the use of a security manager. However, this process is replaced with a two-step process because of the limited resources available on small computing devices.

- The first step is called preverification and occurs outside the small computing device prior to loading the MIDlet. Preverification requires that additional attributes called stack maps are inserted into a class file by software before the second step runs.

- *Stack maps describe the MIDlet's variables and operands located on the interpreter stack.*

- After preverification is completed, the MIDlet class is loaded into the device, and the verifier within the small computing device validates each instruction in the MIDlet class. The MIDlet class is automatically rejected if the verifier detects an error.

- java.lang.Object
- java.lang.String
- java.lang.Thread
- java.lang.Runnable
- java.lang.StringBuffer
- java.lang.Throwable
- **Data Type Classes**
- java.lang.Boolean
- java.lang.Character
- java.lang.Long
- java.lang.Byte
- java.lang.Integer java.lang.Short

- java.util.Enumeration
- java.util.Stack
- java.util.Hashtable
- java.util.Vector
- **Input/Output Classes**
- java.io.ByteArrayInputStream
- java.io.DataOutputStream
- java.io.PrintStream
- java.io.ByteArrayOutputStream
- java.io.InputStream
- java.io.Reader
- java.io.DataInput

- java.io.InputStreamReader
- java.io.Writer
- java.io.DataInputStream
- java.io.OutputStream
- java.io.DataOutput
- java.io.OutputStreamWriter
- Calendar and Time Classes
- java.util.Calendar

J2ME Software Development Kits

- AMIDlet is built using free software packages that are downloadable from the java.sun
  .com web site, although you can purchase third-party development products such as
  Borland JBuilder Mobile Set, Sun One Studio 4 (formerly Forte for Java), and WebGain VisualCafe Enterprise Suite.

- Three software packages need to be downloaded from java.sun.com.

- These are the Java Development Kit (1.3 or greater) (java.sun.com/j2se/downloads.html),
- Connected Limited Device Configuration (CLDC) (java.sun.com/products/cldc/), and
- The Mobile Information Device Profile (MIDP) (java.sun.com/products/midp/).

Also need the J2ME Wireless Toolkit to develop MIDlets for handheld devices (java.sun.com/products/j2mewtoolkit/download.html)

.

- First, install the Java development kit. The Java development kit contains the Java compiler and the jar.exe, which is used to create Java archive files, After downloading the Java development kit package, unzip the package and run the installation program.
- Once the Java development kit is installed, place the c:\jdk\bin directory, or whatever directory you selected for the Java development kit, on the PATH environment variable

- Install the CLDC once the Java development kit is installed. Unzip the downloaded CLDC files from the java.sun.com web site onto the d:\j2me directory (J2ME_HOME) on your computer.
- Need to create the j2me directory if one doesn't exist. Unzipping
- the CLDC package creates the j2me_cldc subdirectory below the j2me directory.
- The j2me_cldc has a bin subdirectory that contains the K Virtual Machine and the preverifier executable files for an assortment of platforms such as win32.
- Each platform is in its own subdirectory under j2me_cldc. Add the j2me\j2me_cldc\bin\win32 subdirectory to the PATH environment variable (see "Setting the Path in Windows" sidebar).
- We should substitute win32 subdirectory with the appropriate subdirectory for your platform.

- Next, download and unzip the MIDP file. Be sure to use \j2me as the directory for the MIDP file. Unzipping the MIDP file creates a midp directory. The name of this directory might vary depending on the version that you download.

- Next, create two environment variables. These are CLASSPATH and MIDP_HOME.
- The CLASSPATH environment variable identifies the path to be searched whenever a class is invoked. The MIDP_HOME environment variable identifies the location of the \lib directory that contains the internal.config file and the system.config file.
- Set the CLASSPATH to d:\j2me\midp1.0.3fcs\classes;.
- Notice that the CLASSPATH terminates with a period. The period implies the current directory and will cause the current directory to be searched if a class is not found in the \j2me\midp1.0.3fcs\classes directory.

Hello World J2ME Style

- We can create first MIDlet once the Java development kit, Connected Limited Device Configuration (CLDC), and Mobile Information Device Profile (MIDP) are installed.
- The HelloWorld MIDlet shows how to create a simple MIDlet that can be invoked directly from the class and from a Java archive file.
- how to create a MIDlet suite that contains two MIDlets. These are HelloWorld and GoodbyeWorld.

HelloWorld.java

- Enter the code into a text editor such as Notepad, and save the file in the j2me\src\greeting directory as HelloWorld.java.

- The HelloWorld MIDlet performs three basic functions in all MIDlets.
- 1. These are to display a text box
- 2. A command on the screen
- 3.Listen to events that occur while the MIDlet is running.
- The HelloWorld MIDlet is created by defining a class called HelloWorld that extends the MIDlet class and implements a CommandListener.
- The HelloWorld class contains three private data members and four methods

- The data members are a Display object, a text box, and a command.
- The methods are startApp(), pauseApp(), and destroyApp(),
- The fourth method is called commandAction() and is invoked by the application manager whenever an event occurs.

- Two packages must be imported at the beginning of the MIDlet to access MIDlet classes and lcdui classes.

- MIDlet classes are screen oriented and create a Display object and then place components of the screen into the Display object.
- The Display object is then invoked later in the MIDlet to display the screen on the small computing device.

- The Display object in this example is called display and will contain a TextBox object called textBox and a Command object called quitCommand. All three objects are private and are defined at the beginning of the HelloWorld class definition.
- The startApp() method contains the necessary statements to invoke previously defined objects.
- The startApp() method begins by creating an instance of the Display object by calling the getDisplay() method. The instance of the Display object is assigned to the display Display object that is previously defined in the class.

- Calling getDisplay multiple times always returns the same Display reference for the specified MIDlet.
- Next, an instance of a command object is created. There are three values required when creating a command object. The first value is the label of the command that will appear on the screen.

`

- The label in this example is Quit. The next value is the type of command, which is a screen command.
- The third parameter determines the priority of the command, which is the first priority—the higher the number, the lower the priority. The application manager uses priority to determine the order in which a command appears in a menu if the MIDlet uses a menu.
- The last instance of an object that is created in the startApp() is a TextBox object. Four values are necessary to create an instance of a TextBox object.

- The first is the caption for the TextBox object followed by the text that will appear in the TextBox object.
- In this example, HelloWorld is the caption and My first MIDlet is the text. The other two values are coordinates used by the application manager to position the TextBox object on the screen.

- Next, the Command object must be associated

  with the TextBox message. This is accomplished by calling the addCommand() method of the TextBox object and passing the addCommand() method the Command object. Once the Command object is associated with the TextBox object, the CommandListener must be associated with the TextBox object in order for the CommandListener to respond to events occurring when the TextBox object is displayed on the screen.

- The setCommandListener() method of the TextBox object is used to associate the TextBox object with the CommandListener.
- And the final statement within the startApp() method associates the TextBox object with the Display object by calling the setCurrent() method of the Display object and passing the setCurrent() method the TextBox object.

- When the application manager of the small computing device runs the HelloWorld MIDlet, the startApp() method is the first method that is invoked, which causes the display that contains the Hello World message and the Quit command to be shown on the screen.
- The HelloWorld MIDlet is required to define a pauseApp() method and a destroyApp() method, but these methods can remain empty because no special action is taken when the HelloWorld MIDLet is paused or destroyed.

- The commandAction() method contains statements that evaluate events that occur while the HelloWorld MIDLet is running.
- The command selected by the user is passed to the commandAction() method as the first parameter. The second parameter is a Displayable object, which is a reference to the TextBox that is associated with the command. A TextBox along with other interface objects are Displayable objects.

- An if statement is used to determine whether the user selected the Command object that is associated with the Hello World TextBox object. If so, the destroyApp() method
- is invoked and is passed a boolean false.
- The destroyApp() method is called before the MIDlet is destroyed; afterwards the notifyDestroyed() method is called to notify the application manager that the HelloWorld MIDLet has entered into the destroyed state.
- Prior to invoking the notifyDestroyed() method, a MIDlet should have completed its own garbage collection.

Compiling Hello World

- Compiling a MIDlet is a two-step process.
- The first step is to use the Java compiler to transform the source file into a class file.
- The second step is to preverify the class file, The preverification generates a modified class file.

- Make j2me\src\greeting the current directory, command at the command line.

- javac -d d:\j2me\tmp_classes -target 1.1 – bootclasspath d:\j2me\midp1.0.3fcs\classes HelloWorld.java

- The compiler produces a file called HelloWorld.class in the j2me\tmp_classes\

  greeting directory. The greeting directory is created because of the package greeting declaration in the source code.

- Next to preverify the HelloWorld.class that was generated by the compiler. Make sure that j2me\src\greeting is the current directory and enter the following command:

- preverify -d d:\j2me\classes -classpath d:\j2me\midp1.0.3fcs\classes d:\j2me\tmp_classes

- We should use two preverify options. The -d option places the class file within the tmp_classes directory.
- The second option is -classpath, which points to the location of the library classes that come with the MIDP. Preverification files are contained in the midp1.0.3fcs\classes directory.
- The output of the javac compiler is in the tmp_classes directory.

Running Hello World

- A MIDlet should be tested in an emulator before being downloaded to a small

  computing device. An *emulator is software that simulates how a MIDlet will run in* a small computing device.
- There are two ways to run a MIDlet.
- These are either by invoking the MIDlet class or by creating a JAR file, then running the MIDlet from the JAR file.

- Make sure that j2me\src\ greeting is the current directory, and then enter the following command.
- midp -classpath d:\j2me\classes greeting.HelloWorld

Deploying Hello World

- A MIDlet should be placed in a MIDlet suite after testing is completed.
- The MIDlet suite is then packaged into a JAR file along with other related files for downloading to a small computing device. This process is commonly referred to as *packaging*.

- In the HelloWorld example, the MIDlet suite contains one MIDlet, which is the HelloWorld.class.
- Before packaging the MIDlet into a JAR file, you'll need to use an editor to create the manifest file shown below.

Helloworld Manifest

- MIDlet-1: HelloWorld, , greeting.HelloWorld
- MIDlet-Name: Hello World
- MIDlet-Version: 1.0
- MIDlet-Vendor: IV IT
- MIDlet-1: HelloWorld, /greeting/myLogo.png, greeting.HelloWorld
- MicroEdition-Configuration: CLDC-1.0
- MicroEdition-Profile: MIDP-1.0

- The manifest describes the JAR file. The manifest file should be saved as manifest.txt in the j2me\src\greeting directory.
- Notice that the MIDlet description within the manifest file contains a graphic call, /greeting/mylogo.png, that is associated with the HelloWorld MIDlet.
- Any PNG-formatted image file can be used in place of mylogo.png.

- You can create the JAR file once the manifest.txt file is saved in the j2me\src\ greeting directory.

- Make sure the j2me\src\greeting directory is the current directory, and then create the JAR file by entering the following command:

- jar -cfvm d:\j2me\midlets\HelloWorld.jar manifest.txt -C d:\j2me\classes greeting

- The final piece of the Hello World package is a JAD file. Create the JAD file shown below using an editor, and save the JAD file in the j2me/src/greeting directory.

Helloworld JAD

- MIDlet-Name: Hello World
- MIDlet-Version: 1.0
- MIDlet-Vendor: IV IT
- MIDlet-Description: My First MIDlet suite
- MIDlet-1: HelloWorld, /greeting/myLogo.png, greeting.HelloWorld
- MIDlet-Jar-URL: HelloWorld.jar
- MIDlet-Jar-Size: 1428

- Copy the HelloWorld.jad file into the j2me/midlets directory, and then make j2me/midlets the current directory
- Invoke the MIDlet by entering the following command.
- midp -classpath HelloWorld.jar -Xdescriptor HelloWorld.jad
- Once you are satisfied that the MIDlet suite packaged in a JAR file is operating properly in the emulator, you can download the JAR file to a small computing device.
- The downloading process is device dependent, and therefore you must refer to the device's documentation or the manufacturer's web site for steps for downloading your JAR file.

What to Do When Your MIDlet Doesn’t Work Properly

- a MIDlet won’t compile or run properly. Although each MIDlet is unique, there are a few common problems that cause a MIDlet to fail. Here are areas to investigate if you experience a failure.

- If the compiler, preverifier, JAR program, or emulator doesn’t run from the command line, review the value of the PATH, CLASSPATH, and MIDP_HOME environment variables to be sure you have included the exact path to these programs.
- Also make sure that the current directory reference (a period) is included in the CLASSPATH

  environment variable.

- Running out of environment space is a common problem on some platforms. We can work around this problem by creating an executable file, such as a batch file in Windows, that sets the environment variables for J2ME components.
- Run this executable file before compiling and testing your MIDlet to temporarily reset environment variables. The environment ariables return to their original values the next time you restart your computer or log in.

- Many types of errors can occur during the compiling and packaging process.
- syntax errors, which you'll be able to fix quickly by reviewing the source code.
- Other errors can be caused by poorly formed command line options and arguments, such as

  failing to insert a space between an option and a period when referencing the current directory.

- Another common occurrence is for a MIDlet suite to run fine in test but fail to run after downloaded to the small computing device. In this case, the application manager on the small computing device might reject the MIDlet suite because the MIDlet suite cannot be run on the device. An oversize MIDlet suite is a likely suspect.

JAD and Manifest(.jad & .mf)

- There will likely be occasions when you need to have your application perform in a certain way, depending on the type of small computing device that runs the application.
- Although you can create versions of your application for specific small computing devices, there is a more efficient approach to tailoring an application to a device.
- First, design your application with switches that activate and/or deactivate routines depending on the value of a setting.
- A setting is a value assigned to a variable that is
- either created within the application or passed to the application as a command line parameter.

Multiple MIDlets in a MIDlet Suite

J2ME Wireless Toolkit

- Building and running a J2ME application at the command line is cumbersome, to say the least, when you are creating a robust application consisting of several MIDlets.
- Creating your application within an integrated development environment is more productive than developing applications by entering commands at the command line.

- The J2MEWireless Toolkit is used to develop and test J2ME applications by selecting a few buttons from a toolbar. However, the J2MEWireless Toolkit is a stripped-down integrated development environment in that it does not include an editor, a full debugger, and other amenities found in a third-party integrated development environment.

MIDlets on the Internet

- The Wireless Toolkit can run MIDlets that access Internet resources by configuring the emulator to interact with a proxy server and let you monitor activities between the MIDlet and the Internet for debugging purposes.
- You configure the emulator for the Internet

  by selecting Edit | Preferences. The Network Configuration tab is used to set the port number and server name of the proxy server.

- The Trace tab is used to set preferencesfor monitoring the interactions between the MIDlet and the Internet.
- There are four options that you can set by selecting the appropriate check boxes
- The Trace Garbage Collection option displays the status of objects that include memory allocation of existing objects, the number of objects on the heap, and the size of the largest free object.
- The status is displayed whenever the garbage collector is
- invoked.
- The Trace Class Loading option will display the name of each class as it is loaded into the emulator.

- The Trace Class Method Calls option logs object and related methods when they are called. Display Exceptions causes all exceptions to be displayed regardless of whether they are caught or uncaught.
- The Performance, Monitor, and Storage tabs are used to fine-tune the Wireless.
- Toolkit for those aspects of an emulator.

- However, J2ME applications are capable of reading the value of a setting from a JAD file and manifest file.
- A setting is a user-defined value created in either file.
- A good practice is to create a user-defined value within the JAD file rather than within the manifest file because the JAD file can be modified without having to repackage your application.
- A manifest file is a component of a package.

- A user-defined value is read by invoking the getAppProperty() method and passing the name of the user-defined value to the getAppProperty() method.
- The getAppProperty() returns the user-defined value from either the manifest file or the JAD file depending on which of these files contains the user-defined value.

Model- Version: M253.

- how to read this user-defined value during run time without having to recompile or repackage the application.
- MIDlet-Name: Best MIDlet
- MIDlet-Version: 2.0
- MIDlet-Vendor: MyCompany
- MIDlet-Jar-URL: http://www.mycompany.com/bestmidlet.jar
- MIDlet-1: BestMIDlet, /images/BestMIDlet.png, Best.BestMIDlet
- Model-Version: M253

- public class BasicMIDletShell extends MIDlet
- public void startApp(){
- {
- System.out.println(getAppProperty("Model-Version"));
- }
- public void pauseApp()
- {
- }
- public void destroyApp(boolean unconditional)
- {
- }
- }

lcdui

- LCDUI is a shorthand way of referring to the MIDP user interface APIs, contained in the javax.microedition.lcdui package. Strictly speaking, LCDUI stands for Liquid Crystal Display User Interface. It's a user interface toolkit for small device screens which are commonly LCD screens.

UNIT-III

- **Commands, Items, and Event Processing:** J2ME User Interfaces, Display Class, The Palm OS Emulator, Command Class, Item Class, Exception Handling. High-Level Display: Screens, Screen Class, Alert Class, Form Class, Item Class, List Class, Test Box Class, Ticker Class. Low-Level Display: Canvas: The Canvas, User interactions, Graphs, Clipping Regions and Animation.

The Reality of Working in a J2ME World

- A small computing device has a radically different hardware configuration than traditional computing devices(desktop computers and servers).
- MoreConcentrate on Hardware Configuration for designing J2ME applications

Traditional and Small Computing Devices

- 1. Traditional- Continues Power- from Power Grid
- Small Computing Devices-Battery Power (Settop Boxes- Mobiles)
- 2. Traditional computing devices and small Computing devices another difference is the network connection.
- mobile small computing devices connect to a network via a radio or infrared connection
- Some nonmobile small computing devices such as set- top boxes use a hard-wired network connection similar to traditional computing devices.

- Data transmission between a mobile small computing device and a traditional computing device is slow in comparison to a hard-wired network connection because radio and infrared technology offers a narrower transmission bandwidth than that found in hard-wired network connections.

- Many users of your J2ME application expect the same response from your application as they experience from desktop computer applications.

- Therefore, you must design your J2ME application to minimize and optimize data transmission with offline data sources.
- One way to optimize your J2ME application is called ROMizing the application for run-time operations. ROMizing creates a machine code image of an application before the application is deployed on the small computing device.
- In comparison, using a justin- time compiler, or other techniques employed by the Java Virtual Machine, optimizes J2SE and J2EE applications.

Best Practices

- *Best practices are proven design and programming techniques used* to build J2ME systems.
- *Patterns are routines that solve common programming problems* that occur in such systems.
- Professional developers use best practices and patterns to avoid making common mistakes when designing and building a J2ME application.

Best Practices and Patterns

- 1. Keep Applications Simple
- 2. Keep Applications Small
- 3. Limit the Use of Memory
- 4. Off-Load Computations to the Server
- 5.Manage Your Application’s Use of a Network Connection

- 6. Simplify the User Interface
- 7. Use Local Variables
- 8. Don’t Concatenate Strings
- 9. Avoid Synchronization
- 10.Thread Group Class Workaround
- 11. Upload Code from the Web Server
- 12. Reading Settings from JAD Files

- 13. Populating Drop-down Boxes
- 14. Minimize Network Traffic
- 15. Dealing with Time
- 16. Automatic Data Synchronization
- 17. Updating Data that Has Changed
- 18. Be Careful of the Content of the startApp() Method.

Keep Applications Simple

- because of limited resources available and the inability to easily expand resources to
- meet application requirements. Typically, you design an application by dividing it into objects that have associated data and methods.(Order Form)

Keep Applications Small

- J2ME application expects the application to download quickly to the small computing device and run among other applications on the device because fewer bytes need to be downloaded and stored in memory on the device.
- In J2ME application should also deploy application as a JAR file.
- A JAR file is a compressed version of a J2ME application.
- On some occasions, even a stripped-down version of application takes too long to download or simply is too large to run on the small computing device.
- In these situations, divide application into several MIDlets, and then combine the MIDlets in a MIDlet suite

Limit the Use of Memory

- In addition to removing unnecessary features from application, design application to manage memory efficiently.
- There are two types of memory management in J2ME application.
- These are overall memory management and peak time memory management.
- Overall memory management is designed to reduce the total memory requirements of an application.
- Peak memory management focuses on minimizing the amount of memory the application uses at times of increased memory usage on the device.

- A primary way to reduce total memory requirements of your application is to avoid using object types. Instead, use scalar types, which use less memory than object types.

- Likewise, always use the minimum data type suited for storing data. For example boolean instead of int will create a dramatic impact on the performance of a J2ME application.

- Peak time memory management requires to manage garbage collection.
- J2ME does have a garbage collector, Here are a few ways to manage own garbage collection:
- First, allocate an object immediately before the object is used in the application rather than at the beginning of application.
- Allocating memory at the beginning of the application reserves memory long before the object will be used within the application. This memory could be utilized by other parts of the application until the application requires the object.
- Next, set all references to objects to null once the application no longer needs the object.
- This decreases the memory application of the object to the minimum memory necessary to store an object reference.

- Always reuse objects instead of creating new objects.

(reduces both memory allocation and the need for processing power).

Memory allocation is reduced because multiple references can use the same object at different times in the application’s life cycle.

- The fewer exceptions that might be thrown, the less memory application requires.
- And the last best practice to reduce memory usage is to release all resources immediately following their use within application.

Off-Load Computations to the Server

- Small computing devices are designed to run applications that do not require intensive processing because processing power common to desktop computers is not available on these devices.
- This means that we must design your J2ME application to perform minimal processing on the small computing device.
- However, the reality is that sophisticated, industrial-strength applications require processing that is beyond the capabilities of these devices.

- But there is an alternative that lets you combine the convenience of a small computing device with an application that requires intense processing.
- i.e client-service J2ME application or web services

Manage Your Application's Use of a Network Connection

- concerned about the availability of a network connection.
- small computing devices are mobile, wireless devices where a network connection is not always available, and even when available, the connection might be broken during transmission due to the positioning of the transmitter and receiver

- Cellular telephone networks use technology that attempts to maintain connection as the mobile device moves from one cell to another cell.
- In reality there are dead zones where the mobile device is outside the range of the cellular telephone transceiver.
- The connection is broken in these dead zones, and sometimes it cannot be automatically reestablished by the telephone company. The drop in communication can occur without warning, as many cellular telephone users have experienced.

- by keeping transmissions short—transfer the minimum information necessary to accomplish a task.
- Instead of retrieving all emails in an inbox, you can retrieve the “From,” “Subject,” and “Data received” fields from the last ten emails that were placed in the inbox. Your J2ME application can present these fields

  on the screen and then give the user the options to select an email to read, select a preview for an email, delete an email, or retrieve the next ten emails.

- Consider using store-forwarding technology and a server-side agent whenever your application requests a lot of information.
- A *server-side agent is software running* on the server that receives a request from a mobile device and then retrieves requested information from a data source, which is very similar to the business logic layer of web services technology.
- The results of the query are then held by the agent until the mobile device asks for the information, at which time the information is forwarded to the mobile device.

- Always build into your mobile application a mechanism for recovering from a transmission drop. For example, retain key information about a request on the mobile device until the request is fulfilled.
- The mobile application can then use the retained key information to resubmit the request either automatically or as a user option if there is a breakdown in communication.

Simplify the User Interface

- Text boxes, combo boxes, radio buttons, check boxes, and push buttons are the Graphical User interface Objects for Desktop Applications.
- small computing devices use a variety
- of user display and input devices cellular telephone, have an inch-square display and a telephone keypad for data input.
- Other devices, such as PDA have wide rectangular screens and a hunt-and-peck keyboard.

- Rule of Thumb
- A,B,C
- Download and Interact
- Depend on Device Manager rather than your own Application
- Use short cut keys than all words

Use Local Variables

- Data storage is a key area within an application for reducing excessive processing.
- Encapsulating data within an object tightly controls access to the data, this advantage is realized at the expense of additional processing time whenever the application accesses the data member.
- Accessing a data member of a class requires more processing steps than accessing the same data if the data is stored as a local variable.
- Therefore, accessing a local variable is less processing intense than accessing a class member.

- We can increase processing of your application if you eliminate the extra steps of accessing a data member of a class by assigning values to local variables.

Don’t Concatenate Strings

- Concatenation also increases the application’s use of memory in addition to increasing the application’s processing requirements, which becomes apparent by comparing processing a string with processing a concatenated string.

- the application wants to compare two strings, both of which are four characters and reside in memory.
- The application instructs the small computing device to copy the first character of each string into the CPU for comparison.
- This process continues until either the null character is reached or a letter pair is different.
- The entire process might require ten reading instructions and five comparison instructions, depending on when a mismatch is discovered.

- additional processing steps are necessary if one of those strings is a concatenated string.
- The concatenation process introduces six additional processing steps: three instructions to read each character of the second string and three more instructions to write those characters to the end of the first string.
- Besides the increase in processing steps, concatenation also requires more memory than if the first string and second string did not have to be concatenated.

- Therefore, you can reduce processing time and memory usage by avoiding concatenating strings.
- An alternative is to concatenate strings before the string is loaded into the small computing device.
- If there is a need to concatenate strings, use a StringBuffer object.
- This makes efficient use of memory when strings are appended to the buffer, although there is additional processing overhead.

Avoid Synchronization

- Invokin a thread is a way of sharing a routine among other operations.
- For example, a sort routine can be shared simultaneously by multiple operations that must sort data(Deadlocks and other conflicts).
- These problems are avoided by synchronizing the invocations of a thread.

- Always use a thread whenever an operation takes longer than a tenth of a second to run because a thread requires less overhead than non-thread invocation methods.
- To increase performance is to avoid using synchronization where possible. Synchronization requires additional processing steps that are not necessary when synchronization is deactivated.

- avoid using synchronization unless there is a high likelihood that conflicts among operations will occur.

Thread Group Class Workaround

- Grouping thread objects is made possible by the ThreadGroup class, but J2ME does not support this class.
- We can work around it, however, by creating your own grouping using the Collection class and store groups of thread objects in a collection and then

  use standard collection methods to start and stop threads in the collection and assign threads to particular thread objects within the collection.

- Less processing is required to assign a thread to an existing thread object than to create a new thread object.
- A common way of reducing the overhead of starting a new thread is to create a group of thread objects that are assigned threads as needed by operations within an application.

Upload Code from the Web Server

- Version management is always a concern of application developers, especially when applications are invoked from within a small computing device.
- It can be a nightmare keeping track of various versions of an application once an application is distributed.

- You can reduce and possibly eliminate problems associated with multiple versions of the same application by requiring invocation of the application from a web server.
- Here's how a small computing device can invoke a web server–based J2ME application:
- midp -transient http://www.mycompany.com/welcome.jad
- Rather than running a local JAD file, the -transient option specifies that the JAD file is located on a web server identified by the URL on the command line.

- In this way, the developer only needs to update one copy of the application, and distribution is handled by making the latest version of the application available on the web server.
- This technique is ideal for set-top boxes that are connected to a web server via a cable television connection or satellite connection.
- Software can be updated each time the settop box comes online without the user or a technician having to reinstall the application.

Reading Settings from JAD Files program reads the value from JAD

```
public class BasicMIDletShell extends MIDlet
{
public void startApp()
{
System.out.println(getAppProperty("Model-Version"));
}
public void pauseApp()
{
}
public void destroyApp( boolean unconditional)
{
}
}
```

JAD file contains User defined data

- MIDlet-Name: Best MIDlet
- MIDlet-Version: 2.0
- MIDlet-Vendor: MyCompany
- MIDlet-Jar-URL: http://www.mycompany.com/bestmidlet.jar
- MIDlet-1: BestMIDlet, /images/BestMIDlet.png, Best.BestMIDlet
- **Model-Version: M253**

- when you need to have your application perform in a certain way, depending on the type of small computing device that runs the application.
- Although you can create versions of your application for specific small computing devices, **there is a more efficient approach to tailoring an application to a device.**
- First, design your application with switches that activate and/or deactivate routines depending on the value of a setting.
- A setting is a value assigned to a variable that is either created within the application or passed to the application as a command line parameter.

- However, J2ME applications are capable of reading the value of a setting from a JAD file and manifest file. A setting is a user-defined value created in either file.
- A good practice is to create a user-defined value within the JAD file rather than within the manifest file because the JAD file can be modified without having to repackage your application.
- A manifest file is a component of a package
- A user-defined value is read by invoking the getAppProperty() method and passing the name of the user-defined value to the getAppProperty() method.
- The getAppProperty() returns the user-defined value from either the manifest file or the JAD file depending on which of these files contains the user-defined value.

Populating Drop-down Boxes

- A drop-down box is a convenient way for users to choose an item from a list of possible items.
- Traditionally, content of a drop-down box is

  loaded from the data source once when the application is invoked and remains in memory

  until the application terminates.

Load the list dynamically from a server whenever the list is long.

- Release the list once the user has made a selection, and then reload the list the next time the drop-down box is invoked.
- In this way, memory used to store the list can be reused between calls to the drop-down box.
- caching a long list limits memory availability to other routines within your application and to other applications running on the small computing device.

Minimize Network Traffic

- Developing a J2ME application is a balancing act between deciding whether processing should be performed by the small computing device or by a server.
- A good practice is to off-load as much processing as is reasonable to a server and minimize the number of processes that need to be invoked by the J2ME application in order to reduce network transmissions.

- Collect all info and forward to server when the process invokes.
- the database server can create the customer list in the desired order without having the user make subsequent requests to manipulate the customer information.

Dealing with Time

- Desktop computers and servers are stationary, and therefore current time reflects the time zone where these devices are located.
- Mobile small computing device because the device can be moved to multiple time zones.
- cellular telephones have a geographic positioning feature that enables the device's operating system to know the exact location of the device.

- Those mobile small computing devices that have a built-in geographic positioning system typically adjust the date/time setting on the device automatically as the device moves to a new time zone.

- The best practice is to always store time based on Greenwich Mean Time (GMT) by using the getTime() method of the Date class. In this way, the time stamp of all the data is recorded in a uniform time zone, facilitating the data analysis.

Automatic Data Synchronization

- Storage of data in a small computing device is temporary because the device usually doesn't have secondary storage.
- All data is stored in primary storage (memory) and can be lost whenever the device loses power.
- Failure to do this will cause both devices to become unsynchronized, resulting in erroneous data being displayed and manipulated by the small computing device.

- A good practice is to build into your J2ME application a routine that automatically uploads the latest data when the J2ME application is invoked.
- Likewise, your J2ME application should automatically download data that has changed to the secondary storage device prior to the termination of the application.

- The small computing device must be connected to the network for both actions to occur. It is common for the device to automatically log onto the network when the device is activated. However, some devices might require the user to log onto the network.
- If the user doesn’t log onto the network, your application is unable to update data in the small computing device with data stored in the secondary storage device.

- A good practice is to prompt the user to open a network connection while your J2ME application begins running or right before the application terminates.
- The prompt should give the user two choices: open a network connection or skip opening a network connection until the next time the J2ME application is opened.
- The prompt should also explain that if the user postpones opening a network connection, the data retained in the small computing device might become outdated and might be lost should the device lose power.

Updating Data that Has Changed

- Keep in mind that synchronizing data can be a time-consuming process, depending on the speed of the network connection and the amount of data that is being updated.
- Data can become outdated in two ways: when data changes on the small computing device and when data changes on the secondary storage device, which is usually the server.

- Three options for updating data: incremental updates, batch updates, and full updates.
- Incremental updates require an exchange of data to occur whenever data changes, either on the small computing device or on the secondary storage device.
- An incremental update is only the changed data exchanged between devices.

- Performance decreases as the number of incremental data changes occur because the changed data is transmitted following the modification of the data.

- The batch update option eliminates the need for incremental updates by updating a batch of data either periodically or on demand, controlled by the user

  of the application.

- A batch update only transmits data that is changed by either the small computing device or the secondary storage device.

- A full update should be available as a user- invoked option because of the time required to update all data.

- Typically, this option is used in an emergency to restore data when incremental and batch updates are unsynchronized.

Be Careful of the Content of the startApp() Method

- The startApp() method is called once during the life of the MIDlet and therefore is a perfect place within your application to store code that is to execute once each time the

  MIDlet is invoked.

  MIDlet is started more than once by

  the device's application manager.

- The application manager might pause the MIDlet while another MIDlet is processing and then restart the MIDlet by calling the startApp() method.
- Statements that should run once during the lifetime of the MIDlet should not be placed in the startApp() method and instead should appear within the MIDlet constructor.

UNIT-IV

Record Management System: Record Storage, Writing and Reading Records, Record Enumeration, Sorting Records, Searching Records, Record Listener.

JDBC Objects: The Concept of JDBC, JDBC Driver Types, JDBC Packages, Overview of the JDBC Process, Database Connection, statement Objects. Result set, Transaction processing, metadata, Data Types, Exceptions.

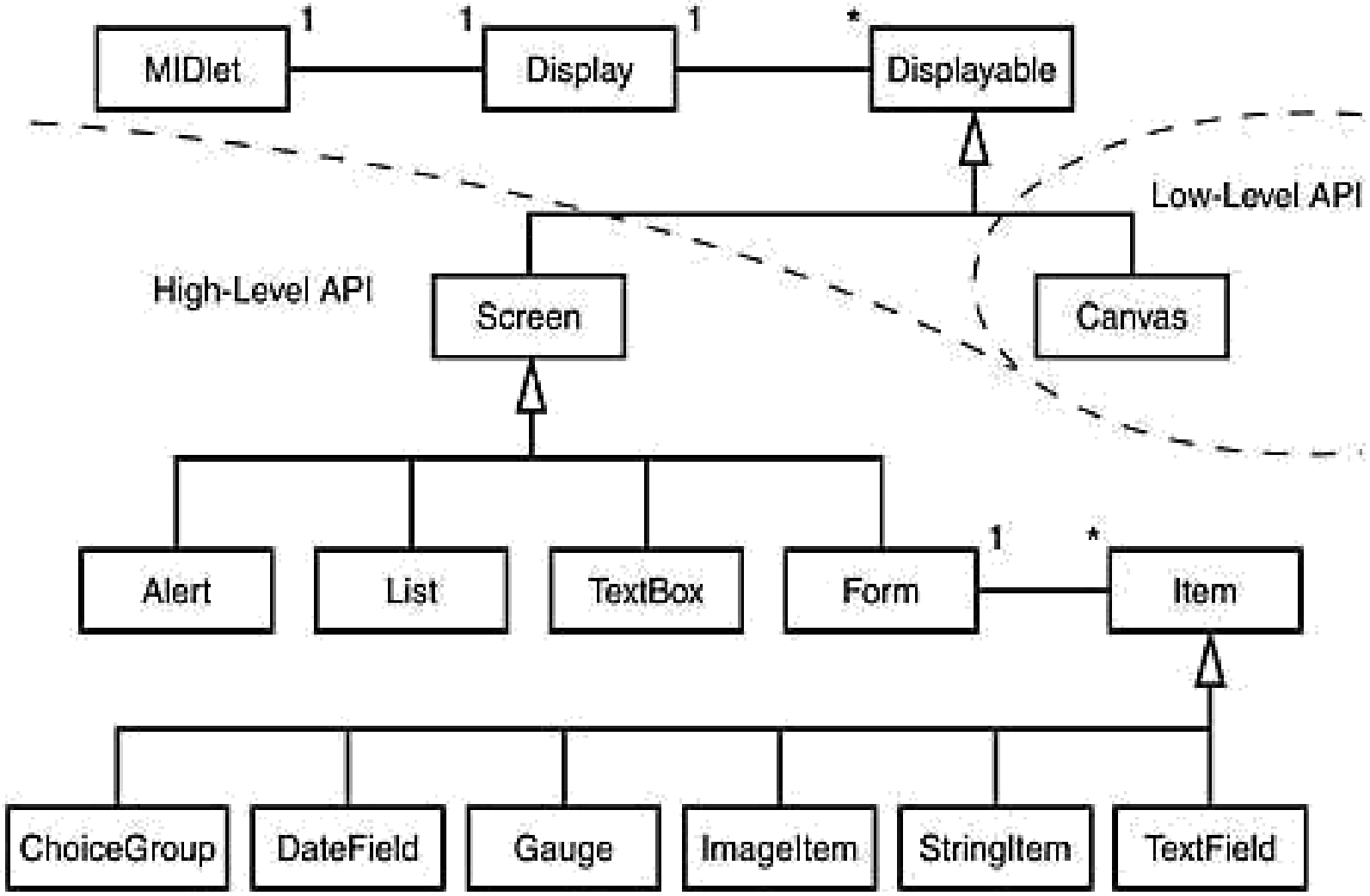
JDBC and Embedded SQL: Model Program, Tables, Indexing, Inserting Data into Tables, Selecting Data from a Table, Metadata, Updating Tablets, Deleting form a Table. Joining Tables, Calculating Data, Grouping and Ordering Data, Sub queries, VIEWS.

- Application uses a variable to accumulate the total of several operations within the MIDlet. Typically, you initialize the variable once when the MIDlet is invoked the first time.
- The initialization must be performed in the MIDlet constructor and not in the startApp() method, otherwise the total will be reset to zero each time the MIDlet is activated after a pause in operations.

MIDP User Interface APIs

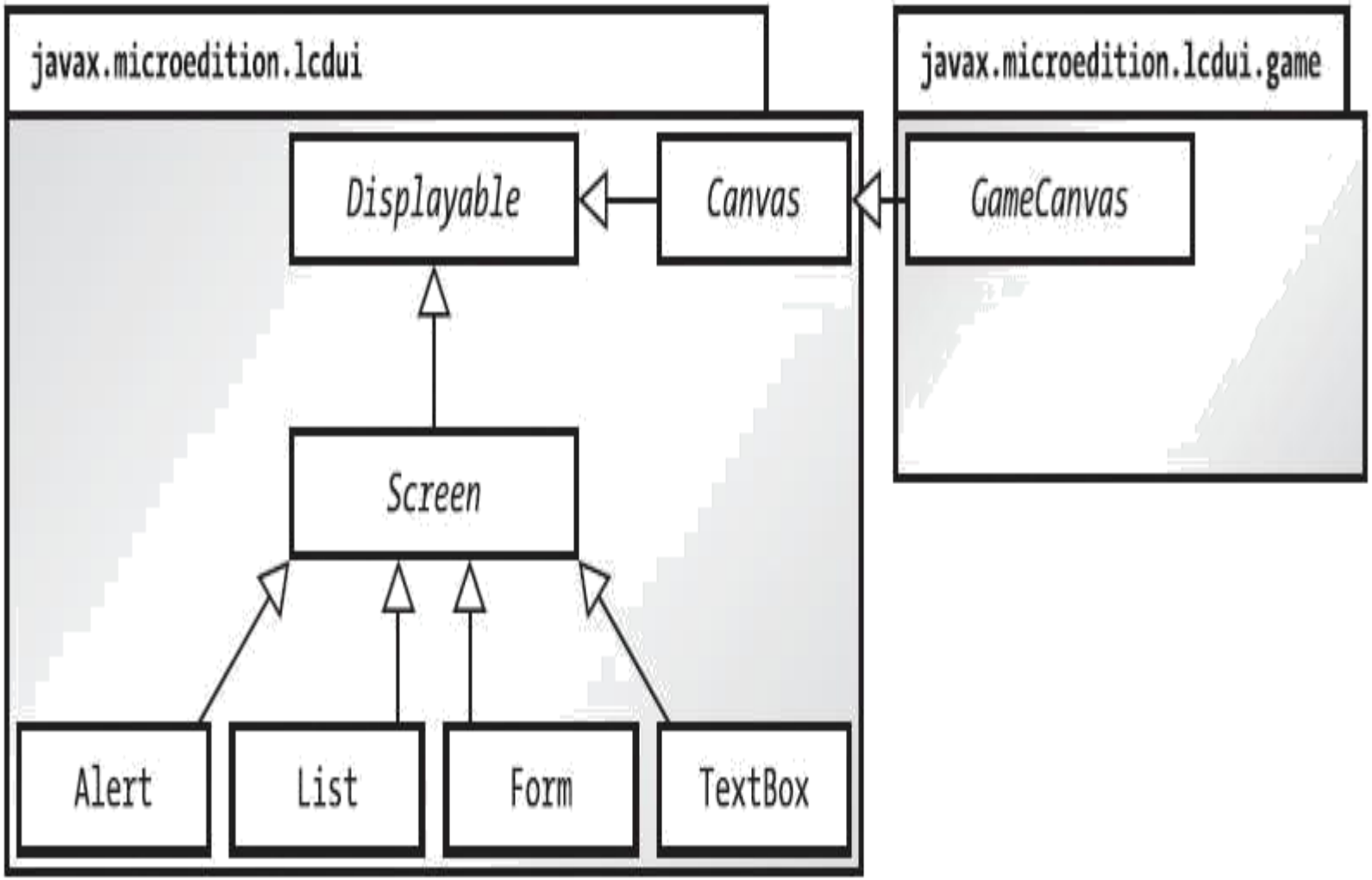
- The MIDP user interface API is divided into a high-and low-level API. The high-level API provides input elements such as text fields, choices, and gauges.
- In contrast to the Abstract Windows Toolkit (AWT), the high-level components cannot be positioned or nested freely.
- There are only two fixed levels: Screens and Items. The Items can be placed in a Form, which is a specialized Screen.

- The high-level Screens and the low-level class Canvas have the common base class Displayable. All subclasses of Displayable fill the whole screen of the device. Subclasses of Displayable can be shown on the device using the setCurrent() method of the Display object.
- The display hardware of a MIDlet can be accessed by calling the static method getDisplay(), where the MIDlet itself is given as parameter.

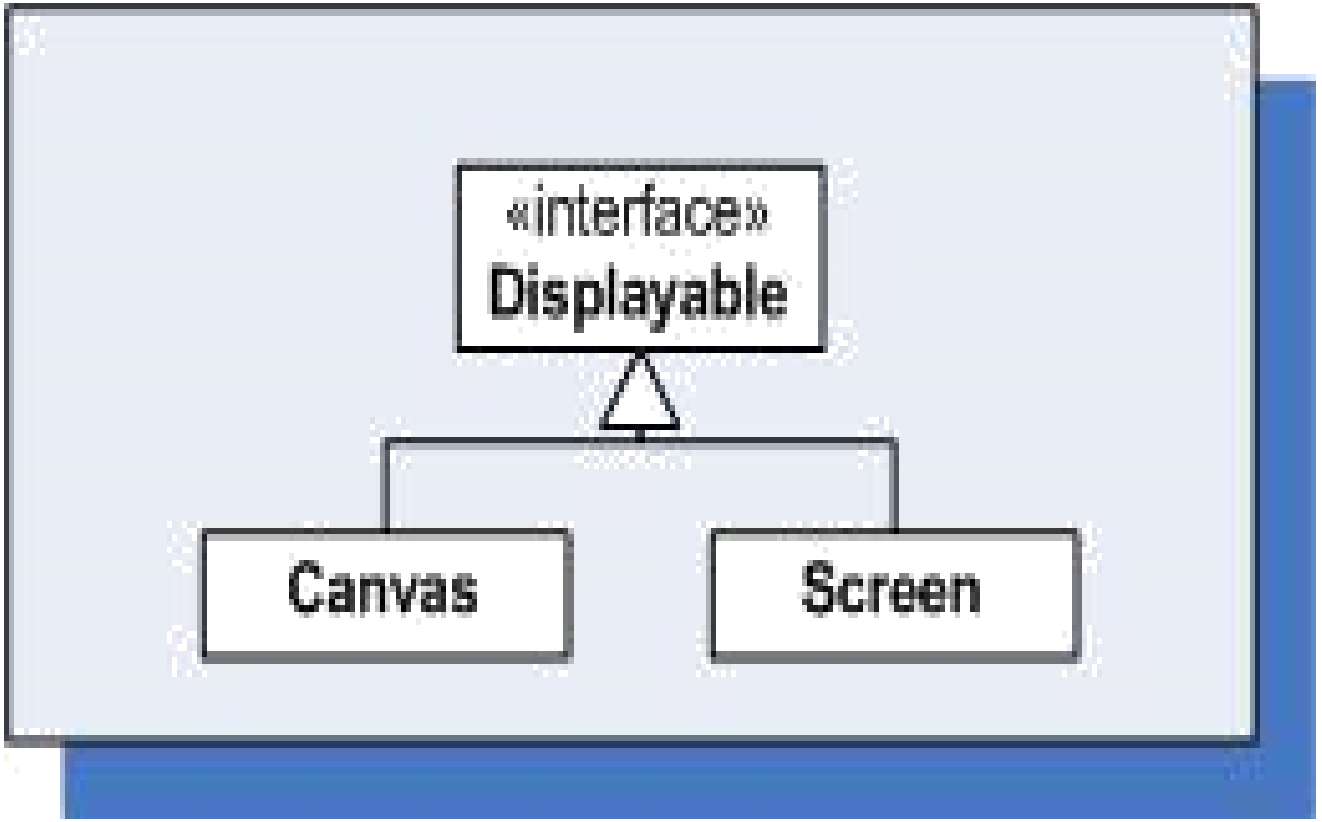

MIDlet
1
1
Display
1
*
Displayable
Low-Level API
High-Level API
Screen
Canvas
Alert
List
TextBox
Form
1
*
Item
ChoiceGroup
DateField
Gauge
ImageItem
StringItem
TextField

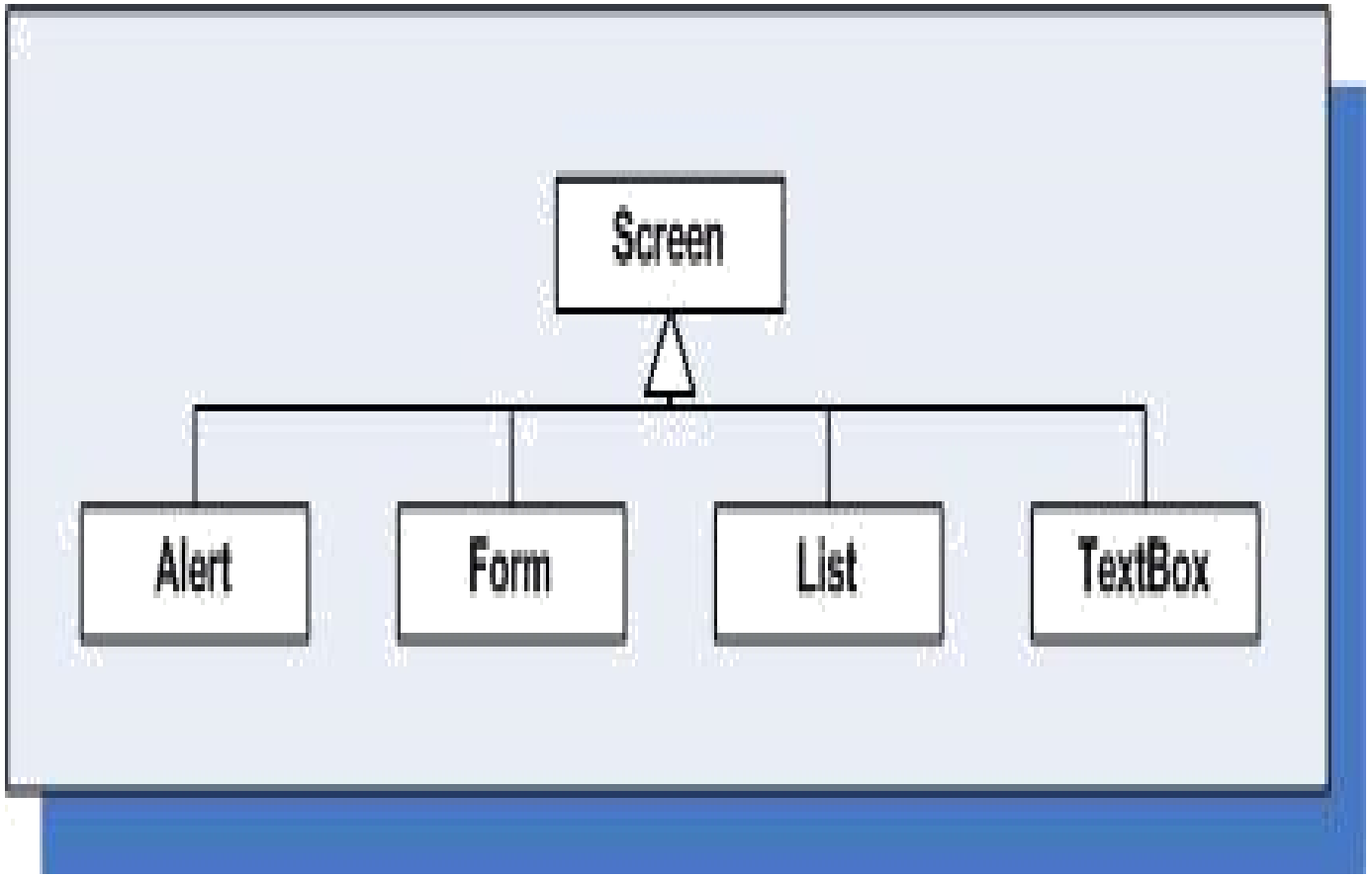
Display and Displayable

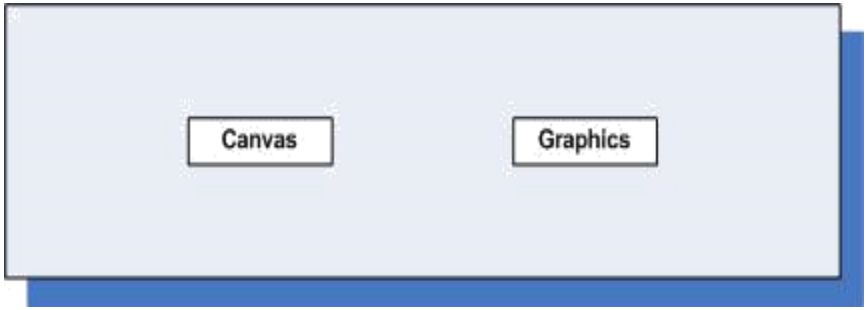
- The difference between Display and Displayable is that the Display class represents the display hardware, whereasDisplayable is something that can be shown on the display. The MIDlet can call the isShown() method of Displayable in order to determine whether the content is really shown on the screen.



Displayables in the javax.microedition.lcdui and javax.microedition.lcdui.game package


«interface»
Displayable
Canvas
Screen


Screen
Alert
Form
List
TextBox


Canvas
Graphics

- A user interface is a set of routines that displays information on the screen, prompts the user to perform a task, and then processes the task.
- Ex: J2ME email application: list of menu options, such as Inbox, Compose, and Exit, and then prompt the user to make a selection by moving
  the cursor keys and pressing a key on the small computing device.

Three kinds of User Interfaces

- 1. Command
- 2. Form
- 3.Canvas

Command User interface

- A command-based user interface consists of instances of the Command class.
- An instance of the Command class is a button that the user presses on the device to enact a specific task.
- For example, Exit, Help etc
- Exit and Help are instance's of the Command

Class's associated with an Exit button and Help button on the keypad to terminate the application and linked the Help key whenever the user requires assistance.

Form User Interface

- A form-based user interface consists of an instance of the Form class that contains instances derived from the Item class such as text boxes, radio buttons, check boxes, lists, and other conventions used to display information on the screen and to collect input from the user.

A form is similar to an HTML form.

Canvas User Interface

- A canvas-based user interface consists of instances of the Canvas class within which the developer creates images such as those used in a game.

Display Class

- Th device's screen is referred to as the display, we can interact with the display by obtaining a reference to an instance of the MIDlet's Display class.
- Each MIDlet has one and only one instance of the Display class, Every J2ME MIDlet that displays anything on the screen must obtain a reference to its Display instance.
- This instance is used to show instances of Displayable class on the screen.

The Displayable class

- The Displayable class has two subclasses.
- 1. Screen class
- 2. Canvas class.

The Screen class

- The Screen class contains a subclass called the Item class, which has its own subclasses used to display information or collect information from a user (such as forms, check boxes, radio buttons).
- The Screen class and its derived classes are referred to as high-level user interface components.

The Canvas class

- The Canvas class is used to display graphical images such as those used for games. Displays created using the Canvas class are considered a low-level user interface and are used whenever you need to display a customized screen.

- Instances of classes derived from the Displayable class are placed on the screen by calling the setCurrent() method of the Display class. The object that is to be displayed is passed to the setCurrent() method as a parameter.
- It is important to note that instances of derived classes of the Item class are not directly displayable and must be contained within an instance of a Form class.

- An instance of an Item class appears on the screen when the setCurrent() method is used to show the form.
- The getCurrent() method of the Display class is used by a MIDlet to retrieve information about the instances of derivatives of the Displayable class.

- Obtain an instance of the Display class by declaring a reference to the instance and then assigning the instance to the reference by invoking the getDisplay() method, Multiple calls to the getDisplay(this) method return the same Display instance.
- **private Display display = Display.getDisplay(this);**

- import javax.microedition.midlet.*;
- import javax.microedition.lcdui.*;
- publi class CheckColor extends MIDlet implements CommandListener {
- private Display display;
- private Form form;
- private TextBox textbox;
- private Command exit;

- The MIDlet is called CheckColor and begins by creating references for instances used in the MIDlet.

  These instances are for the Display class, Form class, TextBox class, and Command class. understand that the instance of the Display class displays a form that contains a text box and the Exit command.
- The color status appears in the text box, and the Exit command terminates the MIDlet.

- public CheckColor()
- {
- display = Display.getDisplay(this);
- exit = new Command("Exit", Command.SCREEN, 1);
- String message=null;
- if (display.isColor())
- {
- message="Color display.";
- }
- else
- {
- message="No color display";
- }
- textbox = new TextBox("Check Colors", message, 17, 0);
- textbox.addCommand(exit);
- textbox.setCommandListener(this);
- }

- Statements within the constructor are executed once during the life of the MIDlet when the MIDlet is invoked.
- Th first statement in the constructor creates an instance of the Display class by calling the getDisplay() method, which is assigned to the display reference.

- Next, an instance of the Command class is created.

- Understand that the label of the instance of the Command class in this example is Exit and the instance is assigned to the exit reference. An instance of the TextBox class is then created.
- The caption of this instance is “Check Colors”
  and is assigned to the form reference.

- The instance of the Command class is then associated with the instance of the TextBox class by calling the addCommand() method and passing the method reference to the

  Command class instance, which in this example is called exit.

- A MIDlet must associate a CommandListener whenever a Command class is instantiated.
- A CommandListener listens for command events to occur during the execution of the MIDlet.
- A command event is the selection of a Command object by the user of the MIDlet.
- Associate a CommandListener with a MIDlet by specifying the listener as an argument to the setCommandListener() method.

- The MIDlet then calls the isColor() method, which returns a boolean value.
- A true indicates that the device can display color.
- A false is returned if the device is incapable of displaying colors.

- The instance of the TextBox class is displayed with a message, depending on the return value of the isColor() method.
- **textbox = new TextBox("Check Colors", message, 17, 0);**
- The first parameter is the caption of the text box, and the second parameter is the text that appears in the text box.

- The Exit command is also associated with the text box, so the user can terminate the MIDlet when the text box appears on the screen.
- Likewise, a CommandListener is also specified for the text box Exit command, which in this case is the MIDlet itself because the MIDlet implements the CommandListener.
- Once the constructor is defined, define the standard methods required by a MIDlet.

```
public void startApp()
    {
display.setCurrent(textbox);
}
public void pauseApp()
{
}
public void destroyApp(boolean unconditional)
{
}
public void commandAction(Command command,
Displayable displayable)
{
if (command == exit)
{
destroyApp(true);
notifyDestroyed();
}
}
}
```

- These are the startApp() method, pauseApp() method, and destroyApp() method.
- The startApp() is called by the device's application manager whenever the MIDlet is started or restarted following a pause in operation.

- The startApp() method contains a statement that calls the setCurrent() method and is passed reference the instance of TextBox class that will be shown on the screen.
- You can include additional statements in the startApp() method as needed by your MIDlet.

- The pauseApp() method definition and the destroyApp() method definition are empty in this example because there are no special statements that must be executed when the MIDlet is paused by the device's application manager.
- The commandAction() method must be defined to receive event reports from the device's application manager.
- Whenever the user selects a command, the commandAction() method is invoked by the application manager to process the command.

- The application manager passes the commandAction() method reference to the selected command, which is then compared to known commands that were created for the MIDlet.
- In this sample the commandAction statement matches the Exit command, which invokes a destroyApp() method to unconditionally terminate the MIDlet.
- After the execution of the destroyApp() method, the notifyDestroyed() method is called to notify the application manager that the MIDlet is terminating.

The Palm OS Emulator

- Before run the Palm OS emulator in the J2ME Wireless ToolkitDownload Palm OS ROM files from the Palm web site (www.palmos.com/dev).
- The ROM file contains the Palm OS required for the emulator to properly perform like a Palm PDA.
- Also need to join the Palm OS Developer Program (free) and agree to the online license (free) for ROM files before you are permitted to download them.
- Be prepared to spend a few minutes downloading since ROM files are fairly large, even when compressed into a zip file.

- Always choose the latest version of the Palm OS for downloading unless you are designing a MIDlet to run on a particular type of Palm device. If your MIDlet is Palm device specific, download the ROM file that corresponds to the Palm OS that runs on that Palm device.

- If download the wrong ROM, because the Palm OS emulator displays an error when running your MIDlet, indicating the proper version of the Palm OS that is required to run your MIDlet on the Palm device that is being tested in the emulator.
- Need to be prompted to enter the location of the ROM file on hard disk into a dialog box the first time that run the Palm OS emulator. Subsequently, the Palm OS emulator uses that ROM file.

Command Class

- We can create an instance of the Command class by using the Command class constructor within your J2ME application.
- The Command class constructor requires three parameters. These are the command label, the command type, and the command priority.
- The Command class constructor returns an instance of the Command class.

- **cancel = new Command("Cancel", Command.CANCEL, 1);**
  - The first parameter of the command declaration is Cancel. Any text can be placed here and will appear on the screen as the label for the command.
- Th second parameter is the predefined command types.
- The last parameter is the priority, which is set to 1. The command created by this declaration is assigned to cancel.

Predefined command types.

Command	Type Description
BACK	Move to the previous screen
CANCEL	Cancel the current operation
EXIT	Terminate the application
HELP	Display help information
ITEM	Map the command to an item on the screen
OK	Positive acknowledgment
SCREEN	No direct key mapping available on device; command will be mapped to object on a form or canvas
STOP	Stop the current operation

- It is important to understand that although a command type is mapped to a key on the device's keypad, the device does not process the command.
- When the user selects the command, the application manager detects the event and passes the selected command to application for processing.

- Priority indicates preference as to the importance of each command object created by your application. Priority is established by the value that assigned to the third parameter of the command declaration.
- A low value has a higher priority than a higher value. The device’s application manager has the option of ignoring the priority or using the priority to resolve conflicts between two commands.

`

- For example, an application manager may use the priority to determine the order in which command labels appear on the screen.
- **A word of caution:** you have no control over how the device's application manager uses the priority of command objects created by your application.

CommandListener

- public void commandAction(Command command, Displayable displayable)
- {
- if (command == cancel)
- {
- destroyApp(false);
- notifyDestroyed();
- }
- }

- Every J2ME application that creates an instance of the Command class must also create an instance that implements the CommandListener interface.
- The CommandListener is notified whenever the user interacts with a command by way of the commandAction() method.

- Classes that implement the CommandListener must implement the commandAction() method, which accepts two parameters.
- The first parameter is a reference to an instance of the Command class, and the other parameter is a reference to the instance of the Displayable class

- The device's application manager calls the commandAction() method and passes the command selected by the user.
- You must evaluate the command to determine the command selected by the user.
- An if statement is used in this example to evaluate the command.

- Compare the command with the reference to the instance of the Command class that was returned when you created the command within your application.
- The commandAction() method must contain all the processing that is to occur when the user selects a command. The destroyApp() method is called to unconditionally terminate the application; and before the application terminates, the notifyDestroyed() method is called to notify the device's application manager that the application is terminating.

Item Class

- The Item class is derived from the Form class, and that gives an instance of the Form class character and functionality by implementing text fields, images, date fields, radio buttons, check boxes, and other features common to most graphical user interfaces.
- The Item class has derivative classes that create those features.

- In many ways, the Item class has similarities to the Command class in that instances of both classes must be declared and then added to the form.
- Likewise, a listener processes instances of both the Item class and the Command class.

- The user interacts with your application by changing the status of instances of derived classes of the Item class, except for instances of the ImageItem class and StringItem class.
- These instances are static and not changeable by the user.
- An instance of the ImageItem class causes an image to appear on the form, and an instance of the StringItem class causes text to be displayed on the form.

- For example application may present options in the form of an instance of the ChoiceGroup class, which is derived from the Item class.
- An instance of a ChoiceGroup class is a check box or radio button. The user makes a selection by choosing a check box or radio button.
- A change in the status of an instance of the Item class is processed by the itemStateChanged() method (defined in the ItemStateListener interface), which is called automatically by the method for an application that utilizes the Item class.
- You must create one itemStateChanged() method for an application that implements an instance of the Item class.

- The itemStateChanged() method is similar to the actionCommand() method used to respond to the invocation of a command by the user of your application.
- The application manager specifically calls the itemStateChanged() method and actionCommand() method independently of each other.
- It is important to understand precisely when the itemStateChanged() method is called because subtle differences in when the method is invoked can alter the way your application reacts to change in the state of an instance of the Item class.

- The state is changed by the user or by your application.
- The change made by the user to a radio button or check box is detected by the listener and causes the device's application manager to call the itemStateChanged() method.
- The state change of the text field by your program doesn't invoke the itemStateChanged() method, although it too is a change of state of an instance of the Item class.

- In contrast, the itemStateChanged() method is invoked if the user changed the content of the text field. The assumption is if the user caused the state to change, then your application needs to consider processing the change in state. However, if your application caused the change, then no additional processing is necessary because the assumption is that any necessary processing would have been completed by your application prior to changing the state.

- The application manager invokes the itemStateChanged() method when the user
- changes focus from the current instance of the Item class to another instance, if the current instance state changed because of user interaction with the instance.
- Th itemStateChanged() method processes the change before focus is set on the other instance.

- In effect, the itemStateChanged() method processes each instance of the Item class as the state is changed by the user.
- Let’s say a form contains a text field and a set of check boxes. The user enters information into the text field and then selects check boxes.
- Between the time the focus leaves the text field and arrives at the check boxes, the device’s application manager calls the itemStateChanged() method, passing it the text of the text field.
- Only after the itemStateChanged() is processed will the user be able to select check boxes.

- **textbox = new TextField("Title", "Text", 4, 0);**

- The text field requires four values. These are title, text, maximum number of characters that can be entered into the text field, and the TextField constraint, which is zero to indicate there isn’t any constraint.

Item Listener

- public void itemStateChanged(Item item)
- {
- if (item == selection)
- {
- StringItem msg = new StringItem("Your color is ",
- radioButtons.getString(radioButtons.getSelectedIndex()));
- form.append(msg);
- }

- Each MIDlet that utilizes instances of the Item class within a form must have an itemStateChanged() method to handle state changes in these instances.
- The itemStateChanged() method contains one parameter, which is an instance of the Item class.
- The instance passed to the itemStateChanged() method is the instance whose state was changed by the user.

- There is one itemStateChanged() per MIDlet, you must include logic within the itemStateChanged() method to identify the Item object that is passed by the device's application manager to the itemStateChanged() method.
- In this example, an if statement is used to compare the incoming instance to one of two instances that the MIDlet created on the form. These instances are a text field and radio buttons.

- First the itemStateChanged() method determines whether the incoming instance is a text field. If so, the MIDlet displays a message that indicates the state of the text field has been changed by the user.

- However, if the incoming instance is not the text field and is a radio button, then a similar statement is displayed at the command line indicating that the user changed the radio button state.
- The device’s application manager must pass the two instances specified in the itemStatechanged() method for a statement to be displayed.

Exception Handling

- The application manager calls the startApp(), pauseApp(), and destroyApp() methods whenever the user or the device requires a MIDlet to begin, pause, or terminate. However, there are times when the disruption of processing by complying with the application manager's request might cause irreparable harm.

- For example, a MIDlet might be in the middle of a communication session or saving persistent data when the destroyApp() method is called by the device's application manager.
- Complying with the request would break off communications or corrupt data.

MIDletStateChangeException

- We can regain a little control of the MIDlet's operation by causing a MIDletStateChangeException to be thrown.
- A MIDletStateChangeException is used to temporarily reject a request from the application manager either to start the MIDlet (startApp()) or to destroy the MIDlet (destroyApp()).

- A MIDletStateChangeException cannot be thrown within the pauseApp() method.

- Many developers place routines that throw a MIDletStateChangeException in the destroyApp() method since terminating a MIDlet during critical processing might have a fatal effect on communication or data.

Throwing a MIDletStateChangeException

- import javax.microedition.midlet.*;
- import javax.microedition.lcdui.*;
- public class ThrowException extends MIDlet
- implements CommandListener
- {
- private Display display;
- private Form form;
- private Command exit;
- private boolean isSafeToQuit;
- //private boolean exitFlag=false;

- public ThrowException()
- {
- isSafeToQuit = false;
- display = Display.getDisplay(this);
- exit = new Command("Exit", Command.SCREEN, 1);
- form = new Form("Throw Exception");
- form.addCommand(exit);
- form.setCommandListener(this);
- }

- public void
  startApp()
- {
- display.setCurrent(form);
- }
- public void pauseApp()
- {
- }

- public void destroyApp(boolean unconditional)
- throws MIDletStateChangeException
- {
- if (unconditional == false)
- {
- throw new MIDletStateChangeException();
- }
- }

- public void commandAction(Command command, Displayable displayable)
- {
- if (command == exit)
- {
- try
- {
- if (isSafeToQuit == false)
- {
- StringItem msg = new StringItem ("Busy", "Please try again.");
- form.append(msg);
- destroyApp(false);
- }
- else
- {
- destroyApp(true);
- notifyDestroyed();
- }
- }

- catch (MIDletStateChangeException exception)
- isSafeToQuit = true;
- {
- }
- }
- }
- }

- Requires the user to select the Exit command twice to terminate the MIDlet .
- When the user selects the Exit command the first time, the device's application manager calls the destroyApp() method where a MIDletStateChangeException is thrown causing the message "Busy Please try again." to be displayed on the screen.
- Th e MIDlet successfully terminates the second time the user selects the Exit button.

- boolean isSafeToQuit variable that is
- used to indicate whether it is safe to terminate the MIDlet.
- In the constructor, the isSafeToQuit is assigned a false, implying that the MIDlet should not be terminated.

- When the MIDlet is loaded into the device, the application manager executes the constructor and calls the startApp() method, where the setCurrent() method is invoked to display the form on the screen.
- The MIDlet then waits for the user to select the Exit command button. When this happens, the CommandListener “hears” the event and calls the commandAction() method, passing the command selected by the user to the method.

- The selected command is then compared to known commands within the MIDlet, which in this example is the Exit command.
- The MIDlet enters the try { } block within the commandAction() method if the Exit command was selected by the user. The value of the exitFlag is then evaluated within the try { } block.
- If the value is false, an instance of the StringItem class is created and is displayed on the screen by passing the instance to the append() method.

- The destroyApp() method is then called and passed a false value. A false value means that there is not an unconditional termination of the MIDlet because processing cannot be disrupted.
- For the sake of this example, we're assuming that processing is ongoing and a fatal error would occur should it not be allowed to complete before the MIDlet is terminated.

- However, the destroyApp() method is passed a true value if the value of the exitFlag is true, indicating that the MIDlet can be terminated unconditionally. Notification of the pending destruction of the MIDlet is then sent by invoking the notifyDestroyed() method.

- Notice that the destroyApp() method is capable of throwing a MIDletState- ChangeException.

  AMIDletStateChangeException is thrown if the destroyApp() method is passed a false value indicating there is a condition to termination of the MIDlet.

  - The MIDletStateChangeException is trapped by the catch { } block in the commandAction() method where the value of the isSafeToQuit is set to true.
  - The next time the user selects the Exit command the destroyApp() is called and passed a true value, meaning the MIDlet can terminate unconditionally.

UNIT-V

- **Generic connection Framework:** The connection, Hypertext Transfer Protocol, Communication Management using HTTP commands, Session Management, Transmit as a Background Process.

- Practically every J2ME application that you develop requires persistence.
- ***Persistence*** is the retention of information during operation of the MIDlet and when it is not running.
- Persistence is common to every Java application written in J2SE, J2EE, or J2ME.
- The manner in which persistence is maintained in a J2ME application differs from persistence in J2SE or J2EE applications because of the limited resources available in small computing devices that run J2ME applications.

- J2ME applications must store information in nonvolatile memory using the Record Management System (RMS). The RMS is an application programming interface that is used to store and manipulate data in a small computing device using a J2ME application.

Record Storage

- Many operating environments contain a file system that is used to store information in nonvolatile resources such as a CD-ROM and disk drive.
- Groups of related information are stored under the same file name. Not all small computing devices have a file system.

- The Record Management System provides a file system–like environment that is used to store and maintain persistence in a small computing device.
- RMS is a combination file system and database management system that enables to store data in columns and rows similar to the organization of data in a table of a database.

- Use RMS to perform the functionality of database management software (DBMS).
- That is, insert records, read records, search for particular records, and sort records stored by the RMS.
- Although RMS provides database functionality, RMS is not a relational database, and therefore cannot use SQL to interact with the data.

- Instead, use the RMS application programming interface and the enumeration application programming interface to sort, search, and otherwise manipulate information stored in persistence.

The Record Store

- RMS stores information in a record store. A *record store compares to a flat file used* for data storage in a traditional file system and to a table of a database.
- A record store contains information referenced by a single name, similar to a flat file and like a table.
- A record store is a collection of records organized as rows (records) and columns (fields).

- Columns contain like data such as first name. Rows contain related data such as a first name, middle name, last name, street, city, state, and postal code.
- RMS automatically assigns to each row a unique integer that identifies the row in the record store, which is called the record ID.
- The record ID is in its own column within the record store.

- The record ID is considered the primary key of the record store. A primary key of the record store serves the same purpose as a primary key in a table of a database, which is to uniquely identify each record in a table.
- Although conceptually envision a record store as rows and columns, technically there are two columns.

- The first column is the record ID, and the other column is an array of bytes that contains the persistent data.

Record Store Scope

- Create multiple record stores as required by MIDlet as long as the name of each record store is unique.
- The name of a record store must be a minimum of one character and not more than 32 characters.
- Characters are Unicode, and the name is case sensitive.

- Record stores can be shared among MIDlets that are within the same MIDlet suite .
- Record stores must be uniquely named within a MIDlet suite, although duplicate names can be used for record stores in other MIDlet suites.

- Let’ say that MIDlet A collects information about customers from a sales representative.
- MIDle B displays customer information collected by MIDlet A.
- MIDlet B can access customer information if both MIDlet A and MIDlet B are in the same MIDlet suite.
- However, MIDlet B is unable to access customer information if MIDlet A and MIDlet B are in different MIDlet suites.

- A system of organizing files in an operating system in which all files are stored in a single directory. In contrast to a hierarchical file system, in which there are directories and subdirectories and different files can have the same name as long as they are stored in different directories, in a flat file system every file must have a different name because there is only one list of files. Early versions of the Macintosh and DOS operating systems used a flat file system. Today's commercial operating systems use a hierarchical file system.

Setting Up a Record Store

- Th openRecordStore() method is called to create a new record store and to open an existing record store.
- This method creates or opens a record store depending on whether the record store already exists within the MIDlet suite.
- Th openRecordStore() method requires two parameters.
- The first parameter is a string containing the name of the record store.

- The second parameter is a boolean value indicating whether the record store should be created if the record store doesn’t exist.

- A true value causes the record store to be created if the record store isn’t in the MIDlet suite and also opens the record store.
- A false value does not create the record store if the record store isn’t located.

- The second version of the openRecordStore() method useful whenever MIDlet tries to open an existing record store.
- Let’s say that MIDlet accesses address information stored in a record store created and maintained by another MIDlet in the same MIDlet suite.

- We don’t want MIDlet to create a new record store if for some reason the record store is unavailable when you tried to open it because another MIDlet creates and maintains the record store.

- Internal resources are utilized to make an open record store available to MIDlets within a MIDlet suite.
- As you know, resources are limited in small computing devices.
- Therefore, you should make a conscious effort not to tie up resources that can be otherwise used for processing by your MIDlet or other MIDlets running on the small computing device.

- To that end, always close any record store that is not in use so that resources utilized by the record store can be reused by other processes.
- You close a record store by calling the closeRecordStore() method.
- The closeRecordStore() method does not require any parameters.

- A record store remains in nonvolatile memory even after the small computing device is powered down. Nonvolatile memory is a scarce resource that needs to be properly managed to ensure that sufficient memory is available when required to store information collected by a MIDlet.

- You can help manage nonvolatile memory by removing all record stores that are no longer being used by MIDlets running on the device. A record store can be deleted by calling the deleteRecordStore() method.
- This method requires one parameter, which is a string containing the name of the record store that is to be removed from the device.

Creating, Opening, Closing, and Removing a Record Store

- To create a new record store, close it, and then remove it from the small computing device. All information contained in the record store is lost when the record store is removed.
- Declare three references for instances of the Display class, Alert class, and RecordStore class.
- The instance of the Display class is required because an alert dialog box is shown should an error be detected by this MIDlet; otherwise, an instance of the Display class is not necessary unless the MIDlet has a user interface.

- A user interface is not required to interact with a record store, although many MIDlets that interact with a record store have a user interface.
- All the actions in this listing occur in the commandAction() method. Typically, routines to create, open, close, and remove a record store are located in appropriate methods throughout the MIDlet.
- Once the instance of the Display class is created, the listing enters the first of three try {} blocks.

- In the first try {} block, the listing attempts to create the record store by calling the openRecordStore() method and passing it the name of the record store and a boolean value.
- The boolean value indicates that the record store should be created if there isn't an existing record store of the same name.
- Errors occurring when creating the record store are trapped by the catch {} block, where an alert dialog box is displayed describing the error.

- Typically, the MIDlet will read from and/or write to the record store at this point in the listing. We'll move on to showing how to close the record store since the purpose of this listing is to provide a framework for working with a record store rather than illustrating how to interact with the record store.
- The closeRecordStore() method is called within the second try {} block to close the record store and release resources used to maintain an open record store.

- Always reopen the record store by calling the openRecordStore() method and passing it the name of the record store that you want to open.

- The catch {} responds to errors that happen if a problem arises when closing the record store. The second catch {} block, similar to other catch
{} blocks in this listing, displays an alert dialog box that contains the error and informs the user that an error occurred.

- Next, the listing determines whether the small computing device contains record stores in nonvolatile memory by calling the listRecordStores() method.
- The listRecordStores() returns a null value if no record stores exist on the device.
- The listing proceeds if at least one record store exists, by entering the third try {} block, where the deleteRecordStore() method is called.

- The deleteRecordStore() method requires one parameter, which is a string containing the name of the record store that is to be removed from the device.

  Any errors occurring during this process are trapped by the catch {} block and displayed in an alert dialog box.

```
import javax.microedition.rms.*;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;
public class RecordStoreExample
extends MIDlet implements CommandListener
{
private Display display;
private Alert alert;
private Form form;
private Command exit;
private Command start;
private RecordStore recordstore = null;
private RecordEnumeration recordenumeration = null;
```

```
public RecordStoreExample
                    ()
{
display = Display.getDisplay(this);
exit = new Command("Exit", Command.SCREEN,
1);
start = new Command("Start", Command.SCREEN,
1);
form = new Form("Record Store");
form.addCommand(exit);
form.addCommand(start);
form.setCommandListener(this);
}
public void startApp()
{
display.setCurrent(form);
}
public void pauseApp()
{
}
```

```
public void destroyApp( boolean unconditional )
    {
}
public void commandAction(Command command, Displayable displayable)
{
if (command == exit)
{
destroyApp(true);
notifyDestroyed();
}
else if (command == start)
{
try
{
recordstore = RecordStore.openRecordStore("myRecordStore",
true );
}
```

```
catch (Exception error)
        {
alert = new Alert("Error Creating", error.toString(),
null, AlertType.WARNING);
alert.setTimeout(Alert.FOREVER);
display.setCurrent(alert);
}
try
{
recordstore.closeRecordStore();
}
catch (Exception error)
{
alert = new Alert("Error Closing",
error.toString(),
null, AlertType.WARNING);
alert.setTimeout(Alert.FOREVER);
display.setCurrent(alert);
}
```

```
if (RecordStore.listRecordStores() != null)
{
try
{
RecordStore.deleteRecordStore("myRecordStore");
}
catch (Exception error)
{
alert = new Alert("Error Removing", error.toString(),
null, AlertType.WARNING);
alert.setTimeout(Alert.FOREVER);
display.setCurrent(alert);
}
}
}
}
}
```

Writing and Reading Records

- Once MIDlet opens a record store, the MIDlet can write records to the record store and read information already stored there using one of two techniques for writing and reading records.
- The first technique is used to write and read a string of data and is used primarily whenever have one data column in the record store such as a list of abbreviations of states.
- The other technique is used to write and read multiple columns of data of different types such as string, integer, and boolean.

- Th addRecord() method is used to write a record to the record store.
- The addRecord() method requires three parameters. The first is a byte array containing the byte value of the string being written to the record store. The second is an integer representing the index of the first byte of the byte array that is to be written to the record store.
- The third is the total number of bytes that is to be written to the record store.

- The first step in writing a string to a record store is to create an instance of a String and assign text to the instance. Next, the string must be converted to a byte array by calling the getBytes() method,
- The getBytes() method returns a byte array.
- **string.getBytes()**

- The second parameter of the addRecord() method is usually zero, and the third parameter is the length of the byte array, indicating that the entire byte array should be written to the record store.
- Typically, information is read from a record store a record at a time and stored in a byte array.
- The byte array is then converted to a string, which is then displayed on the screen or processed further based on the needs of the application.

- MIDlet needs to know the number of records in a record store in order to read all the records from the record store.
- The getNumRecords() method of the RecordStore class returns an integer that represents the total number of records in the record store.
- Use this value as the maximum value for a **for** loop used to step through each record in the record store.

- Call the getRecord() method of the RecordStore class for each iteration of the for loop
- . The getRecord() method returns bytes from the RecordStore, which are stored in a byte array that you create.
- The getRecord() method requires three parameters. The first parameter is the record ID.
- The second parameter is the byte array that you create for storing the record.
- The third parameter is an integer representing the position in the record from which to begin copying into the byte array.

- For example, the following code segment reads the second record from the record store and copies that record, beginning with the first byte of the record, from the record store into the byte array.
- Typically, the first parameter of the getRecord() method is the integer of the for loop, and the third parameter is zero, indicating the entire record is to be copied into the byte array.
- **recordstore.getRecord(2, myByteArray, 0)**

Creating a New Record and Reading an Existing Record

- Many programmers separate routines to write and read records into their own methods.
- These routines are shown in the commandAction() method rather than in separate methods in order to simplify the design and make it easier to understand the technique of writing and reading records.
- Rewrite the program to separate these functionalities once you feel comfortable working with a record store.

- Performs five routines. First a record store is created, and then one record is written to the record store.
- Next, the record is read from the record store and displayed within an alert dialog box.
- Once the user dismisses the alert dialog box, the MIDlet closes

  the record store, then removes the record store from the small computing device.

- The record store is created by calling the openRecordStore() method,
- An exception is thrown if the MIDlet is unable to create the record store, at which time the exception is displayed in an alert dialog box. The MIDlet then enters the routine that writes a record to the record store.

- This routine begins with the creation of a string called “First Record.”
- The string must be stored as a byte array. Therefore, the bytes that make up the string are retrieved by calling the getBytes() method.
- Finally, the addRecord() method is called to write the string to the record store. The addRecord() method is passed three parameters. The first parameter is the byte array that contains the string.

- The second parameter is the index of the first byte, and the third parameter is the total number of bytes that are to be written to the record store.
- The first byte is the first element of the byte array (zero), and the total number of bytes is the number stored in byteOutputData.length. Exceptions thrown when writing the record are displayed in an alert dialog box called within the related catch {} block.

- The next routine reads the record that was written to the record store. The routine
- begins by declaring a byte array that is used to store the bytes read from the record store.
- An integer is also declared and is used when converting the byte array to a string. The routine is written to read all records from the record store, not simply the one record that the MIDlet wrote to the record store in the previous routine, although that is the only record in the record store.

- A for loop is used to step through records in the record store. The maximum iteration of the for loop is set as the return value from the getNumRecords() method, which returns the number of records in the record store.
- The value of the for loop integer (x) represents the record ID. Record IDs begin with one—not zero—therefore the value of the for loop is initialized to one.

- There is always the possibility that the record size exceeds the byte array allocation.
- Avoid this potential problem by evaluating this condition with an if statement, as shown in this routine. If the current record size is greater than the length of the allocated byte array, the MIDlet creates a new byte array the size of the value returned by the getRecordSize() method.

- The getRecord() method is called to retrieve a record from the record store.
- The getRecord() method requires three parameters. The first is the record ID that is being read, which is the current value of the for loop variable.
- The second is the name of the byte array into which the record is copied. The third is the index position of the first byte that is to be copied into the byte array.

- The getRecord() method copies the record from the record store and into the byteInputData byte array and returns an integer representing the length of the record.
- Remember that the record is still in a byte array and must be converted to a string before the record is displayed within the alert dialog box.

- Notice that the message parameter of the Alert method creates a new string using the byte array that contains the record read from the record store.
- Three parameters are necessary to create the string from the context of the byte array.
- The first parameter is reference to the byte array.
- The next two parameters define the first index of the byte array and the last index, which is the number of bytes read by the getRecord() method (length).

- The second parameter is almost always zero, and the third parameter is the length of the record read from the record store, which is the value assigned to the length integer variable.
- The record read from the record store is displayed in the alert dialog box. Once the alert dialog box is dismissed, the MIDlet closes the record store by calling the closeRecordStore() method and then removes the record store by calling the deleteRecordStore() method.

Writing and Reading Mixed Data Types

- It is common for records to consist of mixed data types such as string, boolean, and Integer.
- To save some information like might store the customer name, customer number, and gender.
- A string is used to store a customer name, an integer to store the customer number, and a boolean to indicate gender.

- MIDlet writes a string, an integer, and a boolean value to a record store that is created by the MIDlet. Once the record is written, the MIDlet reads the context of the record,

  which is displayed in an alert dialog box.
- The MIDlet begins by declaring references to objects that are used within the MIDlet.
- Instances of these objects are created within the class constructor. The MIDlet creates a record store called myRecordStore after retrieving reference to the display.

- Any errors occurring while the record store is being created are caught by the catch {} block and displayed in an alert dialog box.
- Th DataOutputStream class has methods that write specific data types to a buffer.
e
- Three of these methods are used in this example. These are writeUTF() method, writeBoolean() method, and writeInt() method. Each is passed the appropriate data.

- The buffered data is placed in the data stream by calling the flush() method.
- The stream is converted to a byte array by calling the toByteArray() method, which returns a reference to the byte array of the stream.
- This reference is passed to the addRecord() method which saves the byte array as a new record in the record store.
- The ByteArrayOutputStream object's internal store is cleared by calling the reset() method.

- Next, the output stream and the data output stream are both closed. Any errors occurring while the data is being written to the record store are trapped by the catch {} block and displayed in an alert dialog box.
- The MIDlet then focuses on retrieving the record from the record store.

- Thi process begins by declaring appropriate references and an array.s

  The array size is set to 100 bytes.

- We must be sure that the size of the array is sufficient to hold all the bytes of the record.

- Reading a mixed data type record from a record store is similar to the routine that writes mixed data types.
- First, you create an instance of the ByteArrayInputStream class.
- The constructor of this class is passed the byte array that was just created.
- We also create an instance of the DataInputStream class and pass reference to the ByteArrayInputStream class to the DataInputStream class constructor.

- The routine used to read records from a record store must assume that more than one record exists, and therefore you need to include a for loop so the MIDlet continues to read records from a record store until the last record is read.

- The reset() method is called to enable reuse of the ByteArrayInputStream's buffer.
- The MIDlet then returns to the top of the for loop and evaluates whether or not to read another record from the record store. If so, the process begins again.

- If not, the inputstream and input data stream are closed, and the contents of the variables are displayed within an alert dialog box. Errors that occur while records are read from the record store are displayed in an alert dialog box.
- The record store is then closed and deleted from the device.

Record Enumeration

- A record store is more like a flat file than a database management system and therefore lacks many sophisticated features that find in a database management system.
- For example,cannot send an SQL query to a record store, nor can ask a record store to search for keywords or sort records, which is commonly performed by a database management system. However, can still perform searches and sorts of records in a record store by using the RecordEnumeration interface.

- An Enumeration provides a way to traverse data elements.
- The Enumeration object manages how data is retrieved from the record store. Changes to the record store are reflected when the record store’s content is iterated.

- Obtain a record enumeration by calling the enumerateRecords() method.

- The enumerateRecords() method requires three parameters.
- The first is the record filter used to exclude records returned from the record store.
- The second is reference to the record comparator, which is a method used to compare records returned from the record store.

- The last parameter is a boolean value indicating whether or not the enumeration is automatically updated when changes are made to the underlying record store.
- The enumerateRecords() method returns a RecordEnumeration, as illustrated in the following code segment.
- There isn’t any filter or comparator method, and the record enumeration is not automatically updated when a change is made to the record store.

- **RecordEnumeration recordEnumeration = recordstore.enumerateRecords(null, null,false);**
  - One of the most common interactions that you'll have with a RecordEnumeration is to step through each record of the RecordEnumeration.
- The **hasNextElement()** method is called to evaluate whether or not there is another record in the RecordEnumeration.
- A boolean true is returned if another record exists; otherwise, a boolean false is returned.

- while (recordEnumeration.hasNextElement())
- {
- //do something
- }

- Retrieve a record from the RecordEnumeration using one of two techniques.
- The first technique is designed to read a record that has a single data type such as a string from the RecordEnumeration.
- The other technique reads a record that has a compound data type.

- **String string = new String(recordEnumeration.nextRecord( ));**

- This code segment calls the nextRecord() method, which returns a copy of the next record in the RecordEnumeration.
- The record is passed to the constructor of the String class and is assigned to the string variable.

- Place this code segment within a conditional loop, such as the while loop to be assured that a record exists in the RecordEnumeration before attempting to copy the record to the string.

- Of course, probably would use an array of strings if a while loop is used; otherwise the MIDlet would be overwriting the previous record assigned to the string, unless plan to process the record within the while loop.

- Move forward or back within the RecordEnumeration by calling either the

nextRecord() method, which moves to the next record, or the previousRecord() method, which moves back one record.

Both the nextRecord() method and the previousRecord() method return a byte array containing a copy of the record.

- You are positioned at the top of the RecordEnumeration when the RecordEnumeration is created. The top is not the first record; you must call the **nextRecord() me**thod to move to the first record. You can move to the last record by calling the **previousRecord()**

  method while at the top of the RecordEnumeration.

  You can return to the top of the RecordEnumeration by calling the **reset()** method.

- Before moving in either direction through the RecordEnumeration, you should always determine whether there are records in the RecordEnumeration, and if so, whether there is a next record or previous record.
- Call the numRecords() method to determine the number of records there are in the RecordEnumeration.
- The numRecords() method returns an integer representing the total number of records.

- If the return value is greater than zero, then evaluate whether there is a next record or previous record depending on the desired direction.
- The hasNextElement() method is called to determine whether there is a next record.
- Call the hasPreviousElement() method to determine whether there is a previous record. Both methods return a boolean value indicating whether or not there is another record.

- You can track your progress through the RecordEnumeration by retrieving the record IDs of records in the RecordEnumeration.
- Let’s say that you determine there are ten records in the RecordEnumeration by calling the numRecords() method.
- The ID of the first record in the RecordEnumeration is zero, and the ID of the last record is nine.

- You can determine the record ID of the next record by calling the nextRecordId() method.
- The nextRecordId() method returns an integer representing the ID of the next record.
- Likewise, you call the previousRecordId() method to retrieve the ID of the previous record.

- Sometimes record IDs can be misleading when the RecordEnumeration is automatically updated whenever a change is made to the underlying record store.

- There are two ways in which automatic updating is activated or deactivated. The first way is when the RecordEnumeration is created.
- As you'll recall, the last parameter in the enumerateRecords() method is a boolean value that if set to true, causes the RecordEnumeration to update automatically.
- The RecordEnumeration is not changed when the underlying record store changes if the boolean value is false.

- The other way to set automatic updating of the RecordEnumeration is by calling the keepUpdated() method.
- The keepUpdated() method has one parameter, which is a boolean value indicating whether or not the RecordEnumeration is automatically updated.

- You can check the status of the automatic updating feature by calling the isKeptUpdated() method.
  - This method returns a boolean value indicating whether or not the RecordEnumeration is automatically updated.
- This is important to know if you are relying on record IDs to plot your way through the RecordEnumeration, because each time a record is inserted or removed from the RecordEnumeration, record IDs are reindexed.

- You can manually cause the RecordEnumeration to be rebuilt by calling the rebuild() method.
- The rebuild() method should be called whenever records in the underlying record store change and the automatic update feature is deactivated.

- Obviously, you'll know when your MIDlet changes the underlying record store and therefore needs to call the rebuild() method.
- However, many times other MIDlets within the same MIDlet suite can also change the record store without notifying your MIDlet.
- In this case, you should create a RecordListener, which notifies your MIDlet that the associated record store has changed and that the MIDlet needs to call the rebuild() method.

- You can call the destroy method to empty the contents of a RecordEnumeration
- and release resources used by the RecordEnumeration.
- This should be done as soon as the MIDlet no longer requires the RecordEnumeration in order to free those resources for other purposes. Remember, a small computing device has limited resources, unlike a PC or server.

Reading a Record of a Simple Data Type into a RecordEnumeration

- 1. Declare references to classes.
- 2. Create instances of classes and assign those instances to references.
- 3. Open a record store and create a new record store if the record store
- doesn’t exist.
- 4. Display any errors that occur when opening/creating a record store.
- 5. Create data in the appropriate data type.
- 6. Convert data to a byte array.
- 7. Write the record to the record store.
- 8. Display any errors that might occur while writing to the record store.
- 9. Create a RecordEnumeration.
- 10. Loop through the RecordEnumeration, copying each record to a variable.
- 11. Display the data in a dialog box.
- 12. Display any errors that occur when reading records from the
- RecordEnumeration.
- 13. Close and remove the RecordEnumeration and the record store.

- The change is in the third try {} block, where records from the record store are copied into the RecordEnumeration and then displayed on the screen.
- The first statement in this try {} block creates a RecordEnumeration by calling the enumerateRecords() method.
- The RecordEnumeration doesn’t use a filter or comparator and is not automatically updated when changes are made to the underlying record store.

- Before entering the while loop, the MIDlet calls the hasNextElement() method to determine whether there is a next record in the RecordEnumeration.
- Remember that the RecordEnumeration is not at the first record, so the hasNextElement() method determines whether there is a record in the record store.
- A boolean true is returned if a record exists, otherwise a false is returned. If a true is returned by the hasNextElement(), the MIDlet copies the first record of the RecordEnumeration into a string, which is then displayed in an alert dialog box.

- Of course, you can further process the record instead of displaying the record, depending on the nature of your application. This process continues until the hasNextElement() method returns a false, indicating no more records exist in the RecordEnumeration.

- The last modification occurs in the fifth try {} block.
- This is where the record store is deleted. The destroy() method is called in this example to
- release resources used by the RecordEnumeration and thereby having the effect of deleting records in the RecordEnumeration.

Reading a Mixed Data Type Record into a RecordEnumeration

- Write and Read mixed data type records.creates an array of strings, integers, and boolean and assigns values to each element of these arrays. Each of these is a record.
- The process of writing mixed data type records to the record store is the same as the process used RecordStore Mixed data types with one small difference.
- Here uses it a loop to write all three records to the record store.

- The next records from the record store are copied into the RecordEnumeration. This process is very similar to the same process that is used to read
- records from the record store, but there are a few subtle differences that need to notice.
- First, the enumerateRecords() method is called to build a RecordEnumeration.
- The enumerateRecords() method doesn't use a filter or comparator and is not updated automatically as changes occur to the record store.

- Next, the hasNextElement() is called to determine whether there is a record in the RecordEnumeration. Statements within the while block are skipped if a false is returned; otherwise, the MIDlet proceeds to the current record by calling the getRecord() method. The getRecord() method requires one parameter, which is the record ID of the record that is being copied into the RecordEnumeration. The record ID is returned by the nextRecordId() method.

- The readUTF() method, readBoolean() method, and readInt() method are then called to return their respective data from the current record of the RecordEnumeration.
- These return values are then concatenated and assigned to a string.
- The string is then displayed in an alert dialog box .
- The same process occurs for each record

  in the RecordEnumeration.
- And releases resources used by the RecordEnumeration

  by calling the destroy() method in the fifth try {} block.

Sorting Records

- Records within a RecordEnumeration are sorted by defining a comparator class that

  is an implementation of the RecordComparator interface.

Within the comparator class define a method that has the logic to compare each record to determine whether

- The compareTo() method returns an integer that is equal to zero, less than zero, or

  greater than zero.

  A zero indicates that both strings are the same. An integer less than zero indicates that the next record precedes the current record in the RecordEnumeration.
- An integer greater than zero indicates that the next record follows the current record in the RecordEnumeration.

- Pass reference to the instance of the RecordComparator() as the second parameter of the enumerateRecords() method.

- The enumerateRecords() then calls the compare() method whenever there is a need to sort records within the RecordEnumerator.
- The direction of the sort is controlled by the logic that you create within the compare() method.

- If you want the sort to appear in ascending order, then return the RecordComparator.PRECEDES when the return value of the compareTo() string is less than the current record and RecordComparator.FOLLOW when the return value is greater than the current record.

- Create a descending sort by reversing these operations:
- return theRecordComparator.FOLLOWwhen the return value of the compareTo() string is

  less than the current record and RecordComparator.PRECEDES when the return value is greater than the current record.

Comparison Values for the compare() Method

- Value Description
- **EQUIVALE NT** Records passed to the compare() method are the same.
- **FOLLOW** The record passed as the first parameter follows the record passed as the second parameter.
- **PRECEDES** The record passed as the first parameter precedes the record passed as the second parameter.

Sorting Single Data Type Records in a RecordEnumeration

- In the third try {} block where an instance of the RecordComparator() interface is created and passed as the second argument to the enumerateRecords() method.
- A null was passed as the second parameter to the enumerateRecords() method because the RecordEnumeration was not sorted.

- The final change occurs at the end of the listing where the compare() method is defined.
- The compare() method receives two parameters called record1 and record2; both are byte arrays supplied by the enumerateRecords() method.

- Those byte arrays are transformed into strings, which are evaluated by the compareTo() method.
- The compareTo() method requires one argument that is the value of the second parameter passed to the compare() method.

Sorting Mixed Data Type Records in a RecordEnumeration

- writes three records containing two columns of data.
- The first column consists of strings and the second column integers.
- The next modification occurs in the third try {} block of where an instance of the RecordComparator interface is referenced and passed as the second parameter to the enumerateRecords() method.

- Another modification is made in the fifth try {} block where a call is made to the compareClose() method of the RecordComparator interface.
- The compareClose() method closes streams opened by the RecordComparator interface to facilitate comparing records.

- The Comparator defines two methods:
- compare() and compareClose().
- The compare() method is called by the

  enumerateRecords() method to compare the next record with the current record within the RecordEnumeration.

Both records are passed as byte arrays to the compare() method.

- Records passed as parameters to the compare() method contain both columns—first a string and then an int.

In real projects you can expect to have multiple columns of different data types.

However, the technique used in this example can easily be expanded to handle records of any number of columns and data types.

- The compare() method begins by creating a byte array input stream from the first record and then using the byte array input stream to create a data input stream.
- Each column in the record is read using the appropriate read*XXX() method, as* described previously.

- It is important to remember to read each column
- in the order in which the column appears in the record. In this example, the first
- column is a string, and the second column is an integer. Therefore, the readUTF()
- method must be called before calling the readInt() method. This is also true if the
- record has other columns.

- Only the return value of the readInt() method is assigned to a variable in this example
- because the value of the second column is being used to sort the record. The return value
- of the readUTF() method is not retained because the value of this column is not being
- sorted. You don’t need to read columns of a record beyond the column that is the key
- to the sort because only the column with the sort key is processed.

- The same process is used to extract the column used as the sort key for the second record passed to the compare() method. The integer columns from both records are then compared, and the appropriate comparison value is returned by the compare()Method.

Searching records

- Searching is referred to as *filtering*, *where the filter is defined by the search criteria*.
- Records that match the search criteria are copied into the RecordEnumeration.
- Those not matching the search criteria are filtered from the RecordEnumeration.
- Searching for a record in a record store is very similar to sorting records in that you define an implementation of an interface.
- In this case the implementation you define filters records contained in a record store rather than sorting records in a RecordEnumeration.

- Th RecordFilter interface is used when searching for a record.

- You must define two methods when defining an implementation of the RecordFilter interface.

- These are the matches() method and the filterClose() method.

- The constructor accepts the search criteria as a parameter when your MIDlet creates an instance of the implementation class.
- The matches() method contains the logic

  necessary to determine whether a column fits the search criteria and returns a boolean value indicating whether or not there is a match.

  The filterClose() method frees resources used by the implementation of the RecordFilter interface once the search is completed.

- Logic contained in the matches() method reads one or multiple columns from the current record and then applies logical operators to determine whether the record meets the search criteria.
- You determine the logic used to decide whether or not a record should or should not be included in the RecordEnumeration.
- Furthermore,you can sort the filtered records by first searching for a subset of records in the record store, then sorting those records.

- The next two sections illustrate how to search a record store that contains records of a single data type and records containing multiple data types.

- Listing 8-15 shows how to search records that contain a single data type (Figure 8-9).
- Listing 8-16 is the JAD file for Listing 8-15. You'll notice that Listing 8-15 is a modified
- version of Listing 8-13. Both write three records to a record store. Each record listings
- has one column that is a String data type. Listing 8-15 searches for one of those records
- and displays the results of the search in an alert dialog box.

- You'll notice in the third try {} block that a statement creating an instance of the Filter replaces the statement that creates an instance of the Comparator.
- The Filter is the implementation of the RecordFilter that is defined at the end of the listing.
- The constructor of the Filter is passed the word "Bob," which is the search criteria.

- The matches() method is automatically called to filter records that don’t meet the search criteria.
- The matches() method requires one parameter, which is the record that is being matched to the search criteria.
- The record is an array of bytes passed by the
- enumerateRecords() method.
- The record is then converted to lowercase characters and assigned to a String variable.