# LECTURE NOTES

## ON

# MOBILE APPLICATION DEVELOPMENT

**BCA VI semester**

**Mr. D RAHUL**

Assistant Professor

# UNIT-I

## J2ME OVERVIEW:

- the Java development team enhanced the capabilities of Java to dramatically reduce the complexity of developing a multi-tier application.
- The team grouped features of Java into three editions, each having a software development kit (SDK).
- The original edition of Java, called the Java 2 Standard Edition (J2SE), consists of application programming interfaces (APIs) needed to build a Java application or applet.
- The Java 2 Micro Edition (J2ME) contains the API used to create applications for small computing devices, including wireless Java applications.
- The development team at Sun worked on Java in the early 1990s to address the programming needs of the fledgling embedded computer market, but that effort was sidetracked by more compelling opportunities presented by the Internet.
- As those opportunities were addressed, a new breed of portable communications devices opened other opportunities at the turn of the century. Cell phones expanded from voice communications devices to voice and text communications devices.
- Pocket electronic telephone directories evolved into personal digital assistants. Chipmakers were releasing new products at this time that were designed to transfer computing power from a desktop computer into mobile small computers that controlled gas pumps, cable television boxes, and an assortment of other appliances.
- J2ME is a reduced version of the Java API and Java Virtual Machine that is designed to operate within the sparse resources available in the new breed of embedded computers and microcomputers.

## Inside J2ME:

- Consumers expect the same software and capabilities found on their desktop and laptop computers to be available on their cell phones and personal digital assistants.
- Developers seeking to build applications that run on cell phones, personal digital assistants, and various consumer and industrial appliances must strike a balance between a thick client and a thin client.
- A **thick client** is front-end software that contains the logic to handle a sizable amount of data processing for the system.
- A **thin client** is front-end software that depends on back-end software for much of the system processing.
- Processing on the wireless device might involve two steps: First the software performs a simple validation process to assure that all fields on the form contain information.
- Next the order is transmitted to the back-end system.

- The back-end system handles adjusting account balances and other steps involved in processing the order.
- A confirmation notice is returned by the back-end system to the wireless device, which displays the confirmation notice on the screen.
- A key benefit of using J2ME is that J2ME is compatible with all Java-enabled devices. A Java-enabled device is any computer that runs the Java Virtual Machine.

## How J2ME Is Organized:

- Traditional computing devices use fairly standard hardware configurations such as a display, keyboard, mouse, and large amounts of memory and permanent storage.
- The Java Community Process Program has used a twofold approach to addressing the needs of small computing devices.
- First, they defined the Java run-time environment and core classes that operate on each device. This is referred to as the *configuration*.
- A configuration defines the Java Virtual Machine for a particular small computing device.
- There are two configurations, one for handheld devices and the other for plug-in devices.
- Next, the Java Community Process Program defined a profile for categories of small computing devices.
- A *profile* consists of classes that enable developers to implement features found on a related group of small computing devices.

## J2ME Configurations:

- **Connected Limited Device Configuration (CLDC)**
  - The **CLDC** is designed for 16-bit or 32-bit small computing devices with limited amounts of memory.
  - **CLDC** devices usually have between 160KB and 512KB of available memory and are battery powered.
  - CLDC devices use the K Java Virtual Machine (KVM) implementation, which is a stripped-down version of the JVM.
  - CLDC devices include pagers, personal digital assistants, cell phones, dedicated terminals, and handheld consumer devices with between 128KB and 512KB of memory.
- **Connected Device Configuration (CDC).**
  - CDC devices use a 32-bit architecture, have at least two megabytes of memory available, and implement a complete functional JVM.
  - CDC devices include digital set-top boxes, home appliances, navigation systems, point-of-sale terminals, and smart phones.

## J2ME Profiles:

- A profile consists of Java classes that enable implementation of features for either a particular small computing device or for a class of small computing devices.
- Small computing technology continues to evolve, and with that, there is an ongoing process of defining J2ME profiles.
- Seven profiles have been defined as of this writing. **These are**
- **the Foundation Profile**(CDC)
- **Game Profile**(CDC)
- **Mobile Information Device Profile**(CLDC)
- **PDA Profile**(CLDC)
- **Personal Profile**(CDC)
- **Personal Basis Profile**(CDC)
- **RMI Profile.** (CDC)

## J2ME and Wireless Devices:

- Developers, mobile communications device manufacturers, and mobile network providers are anxious to fill this need, but there is a serious hurdle: mobile communications devices utilize a number of different application platforms and operating systems.
- Without tweaking the code, an application written for one device cannot run on another device.
- The Wireless Application Protocol (WAP) forum became the initial industry group that set out to create standards for wireless technology.
- The WAP forum created mobile communications device standards referred to as theWAP standard.
- TheWAP standard is an enhancement of HTML, XML, and TCP/IP. One element of this standard is the Wireless Markup Language specification, which consists of a blend of HTML and XML and is used by developers to create documents that can be displayed by a **micro browser**.
- A **micro browser** is a diminutive web browser that operates on a mobile communications device.
- The WAP standard also includes specifications for a **Wireless Telephony Application Interface (WTAI)** specification and the **WML Script** specification.
- **WTAI** is used to create an interface for applications that run on a mobile communications device.
- **WML** Script is a stripped-down version of JavaScript.

- WAP forum provided the framework within which developers can build mobile communications device applications, they still had to overcome a common hurdle found in every rapidly developing technology.
- J2ME applications referred to as a **MIDlet** can run on practically any mobile communications device that implements a JVM and MIDP.

# Small Computing Technology

## Wireless Technology:

- Wireless technology that is used in small computing devices and mobile communications devices is the same radio technology Guglielmo Marconi used to provide an alternative communication means to the telegraph and the telephone.
- Radio technology is based on the wave phenomenon.
- A wave is a characteristic of vibrating molecules, which you see whenever you move a knife up and down in the still water of a dishpan.
- Waves are measured in two ways: by the wave height and by the wave frequency.
- The **wave height** is referred to as the wave's amplitude.
- The **frequency** of the wave is simply called frequency, which is measured as the number of waves per second.
- **Low-frequency** wave called a sound wave produces a frequency that can be heard by humans. Sound waves travel a short distance through air.
- A **higher-frequency** wave called a radio wave cannot be heard but can travel long distances in all directions and through solid objects.
- And even higher frequencies called light waves take on other characteristics.
- **Light waves** can be seen, travel a long distance in a limited direction, and cannot penetrate solid objects.
- Waves are grouped according to frequencies that have similar characteristics in the **electromagnetic spectrum**.
- The **radio spectrum** has divisions for television, microwave, and X-ray frequencies.
- The **light spectrum** has divisions for infrared light, visible light, and ultraviolet light.
- Radio signals are transmitted in the frequency range from 10 kilohertz to 300,000 megahertz.
- A hertz is one wave per second, kilohertz is 1,000 waves per second, and a megahertz is a million waves per second.

## Radio Transmission:

- Radio transmission consists of two components. These are a transmitter and a receiver, both of which must be tuned to the same frequency. A transmitter broadcasts a steady wave called a carrier signal that does not contain any information.
- A carrier signal has two purposes. First, the carrier signal establishes a communications channel with the receiver (Figure 2-5). The receiver knows the channel is open when the carrier signal is detected. The carrier signal also serves as the wave that is encoded with information during transmission.

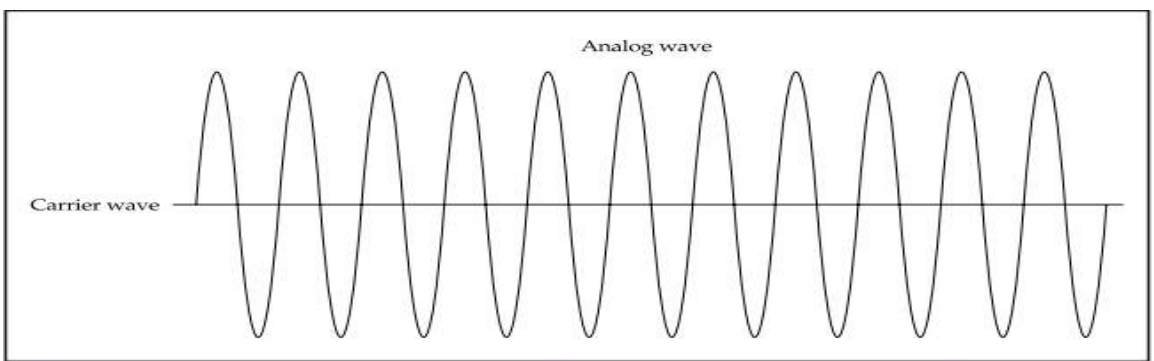



**Figure 2-4.** *A carrier signal is a broadcast wave that does not contain any information.*

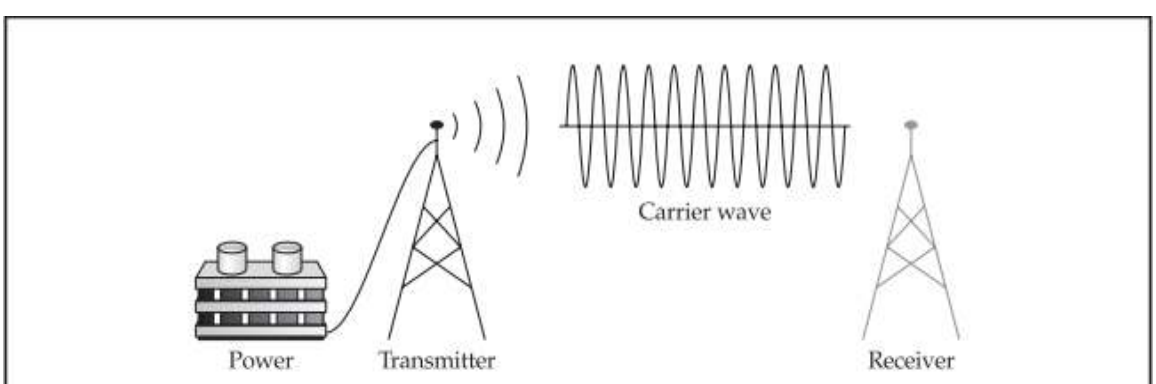



**Figure 2-5.** *A carrier signal is used to establish a communications channel.*

- A radio transmitter encodes patterns of sound waves detected by a microphone by modifying the carrier signal wave (Figure 2-6). The receiver decodes the pattern from the carrier wave and translates the pattern into electrical current that directs a speaker to regenerate the sound waves captured by the microphone attached to the transmitter.

## Limitations of Radio Transmissions

- The distance a radio signal travels is based on the amount of energy used to transmit the radio wave.
- Radio waves are measured in watts. A radio signal transmitted at 50 megawatts travels twice the distance a 25-megawatt radio signal travels.
- A radio signal gradually loses power the farther it travels away from the transmitter. Radio engineers extend the range of transmission by using a repeater.
- A repeater (Figure 2-7) is both a radio receiver and radio transmitter, also known as a transceiver.
- A repeater receives a radio signal and then retransmits the signal, thereby increasing the distance the signal travels. Retransmission introduces new energy to power the signal for longer distances.

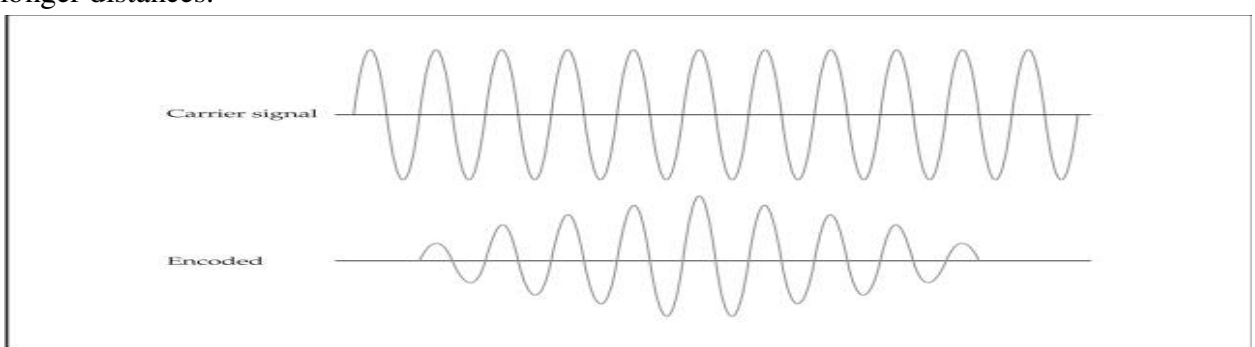



**Figure 2-6.** *Sound waves detected by a microphone modify the carrier signal.*

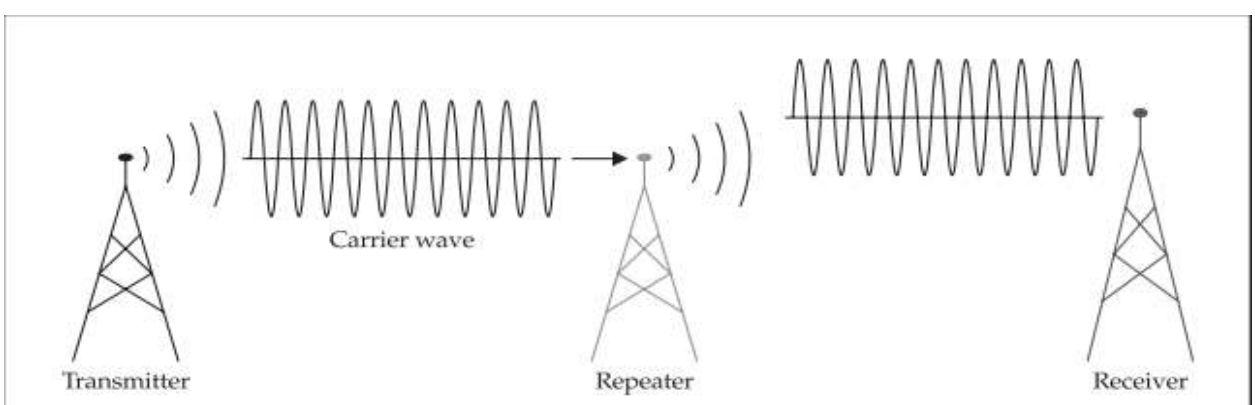



**Figure 2-7.** *A repeater receives and retransmits a signal.*

## Radio Data Networks:

- Radio transmissions are commonly used to broadcast analog voice information on radio waves that travel 360 degrees over the air and through many physical obstructions.

- Information is traditionally encoded as variations of an aspect of the wave. Encoding is achieved by modifying the amplitude of the wave, known as amplitude modulation (AM), or modifying the frequency of the wave, called frequency modulation (FM).
- Encoding uses many values to represent information using AM and FM. Hundreds of thousands of radio waves are simultaneously and independently transmitted.
- Sometimes a radio receiver picks up an erroneous radio signal while tuned to its primary frequency.
- The erroneous radio signal is known as interference and can disrupt the accurate decoding of the transmitted information.
- Digitizing information enables receivers to accurately decode transmitted information because the degree of uncertainty in the encoded information is far less than experienced in analog encoded information.
- Both an analog signal and a digital signal are waves. They differ by the way information is encoded into the signal. Information is represented in an analog signal as many values.
- There are three types of wireless radio networks: low-power single frequency, highpower single frequency, and spread spectrum.
- **Low-power single frequency** covers an area of 30 meters, which is about the area of a small building such as a warehouse or a stock exchange trading floor.
- **A high-power single frequency** wireless radio network can cover a metropolitan area. Both low-power single frequency and high-power single frequency radio networks are exposed to the same security risk.
- **A spread-spectrum** wireless radio network uses multiple frequencies to transmit a signal using either **direct sequence modulation or frequency hopping.**
- **Direct sequence modulation** breaks down information into parts and then simultaneously transmits each part over a different frequency. The receiver must tune to each frequency to receive each part, then reassemble parts into the full message.
- **Frequency hopping** transmits information rotating among a set of frequencies. The receiver must be tuned to each frequency according to the transmission rotation.

## Data Packets:

- Radio transmitters send one message at a time over a communications channel.
- Digital radio networks use packet switching technology to transmit multiple messages simultaneously over a communications channel.
- Each message is divided into small pieces and placed in an electronic envelope called a packet (Figure 2-8).
- A **packet** contains information that identifies the sender and the receiver, a digitized portion of the message, the sequence number of the packet, and error-checking information.

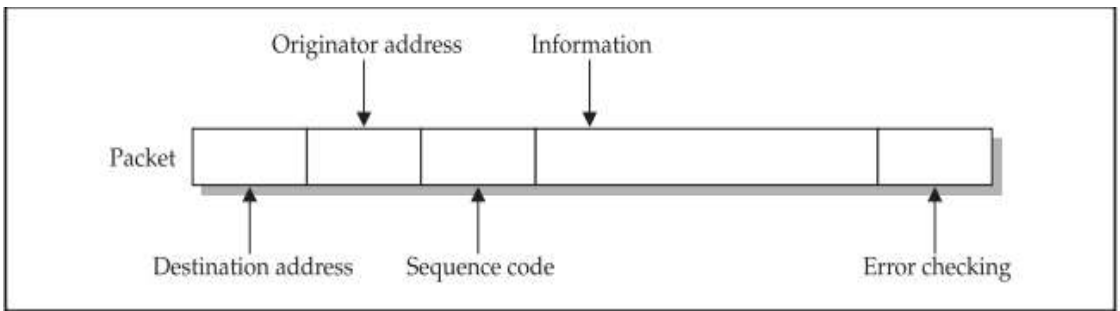



**Figure 2-8.** *A message is divided into small pieces that are placed in an electronic envelope called a packet.*

- To reassemble packets, the receiver uses the packet sequence number.
- A **transmitter** continuously sends packets from multiple messages over a communications channel. Packet switching technology is more efficient than traditional transmission methods because packet switching utilizes pauses in a transmission to send packets.
- A transmission pause is caused when a message isn't ready for transmission.
- **Software** running on the transmitter manages multiple outgoing messages to assure that each message is divided and placed into packets and the packets are transmitted.

## Microwave Technology:

- Microwave is a subspectrum of the radio spectrum and has many characteristics of radio waves.
- microwaves travel in one unobstructed direction. Any obstruction, such as a mountain or building, disrupts microwave transmission.
- There are **two** kinds of **microwave networks**: **terrestrial** and **satellites.**
- **Terrestrial microwave networks** transmit a microwave signal over a terrain, such as buildings in an office complex.
- **Satellite microwave networks** transmit a microwave signal between a ground station and orbiting satellites and among orbiting satellites.
- Earth-to-satellite transmissions are slower than terrestrial microwave transmissions, which causes unnatural pauses to occur in the transmission.

## Satellite Networks:

- A satellite is an orbiting repeater that receives a microwave transmission from an earth station or from other satellites, then retransmits the signal to a microwave receiver located on the ground or in another satellite.
- The first generation of satellites used for the military were stationed in **geosynchronous orbit** at a fixed location 22,300 miles above the surface of the earth.

- The geosynchronous orbit hampers real-time transmission because of the signal delay between earth and the satellite, which makes geosynchronous orbiting satellites unacceptable for commercial two-way real-time communication.
- A newer breed of satellite technology, called **Low Earth Orbiting Satellite (LEOS),** overcame the communications delay by positioning satellites lower than geosynchronous orbit—**between 435 miles and 1,500 miles** above the earth.
- LEOS eliminated delays in communication, but introduced two new problems.
- First, LEOS covers a smaller area of the earth, and therefore more satellites are required to cover the same ground area as covered by geosynchronous satellites. The other problem is the orbital speed.
- LEOS travels faster than the earth's rotation and requires ground stations to locate LEOS before beginning transmission. Geosynchronous satellites always remain in the same position above the ground station.
- In an effort to compromise between LEOS and geosynchronous satellites, another breed of satellites called the **Middle Earth Orbit (MEO)** was developed. MEO orbits between LEOS and geosynchronous satellites—6,000 to 13,000 miles—and thus has less delay than geosynchronous satellites and poses less difficulty than LEOS for ground stations to locate.

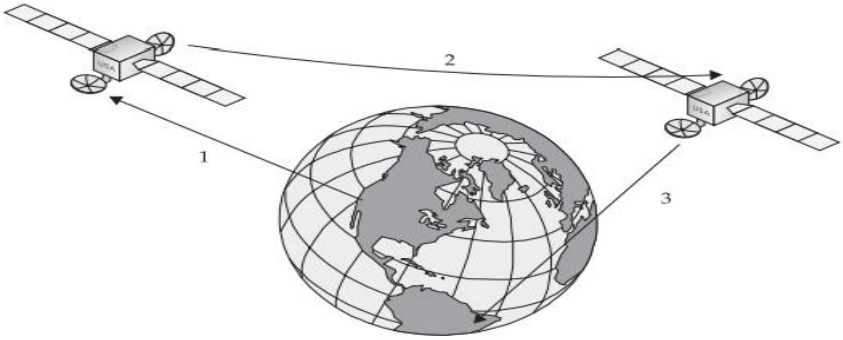



**Figure 2-9.** *A microwave signal is used to communicate among orbiting satellites and ground stations.*

## Mobile Radio Networks:

- The infrastructure of cellular telephone technology is the backbone of wireless small computing mobile communications and enables these devices to connect to traditional communications systems.
- The forerunner of cellular telephone technology is a private radio technology. Private radio transmitted analog information when first introduced but later expanded into digital communication as the need for paging and messaging services materialized.
- Companies can operate their own private radio network by acquiring broadcast rights to a specified radio frequency from the Federal Communications Commission and purchasing the necessary broadcast equipment.

- Alternatively, companies can lease broadcast time from organizations that offer **Specialized Mobile Radio (SMR) network services**.

## Cellular Telephone Networks:

- A cellular telephone network comprises mobile transceivers, called cellular telephones, and a network of fixed transceivers, called base stations, that are strategically positioned along the terrain (Figure 2-10).
- Base stations are used to connect cellular telephones to the ground-based telephone system. There are **two** kinds of **cellular networks**: **analog and digital**.
- Cellular telephones used analog technology at that time. This changed in mid-1995 when IBM developed technology that digitized information transmitted over the cellular telephone network.
- A cellular telephone is in continuous communication with base stations as it moves throughout the cellular network. Transmission from a cellular telephone is broadcast 360 degrees and is received by a base station closest to the cellular telephone.
- Cellular telephone networks are designed so that the signal is automatically transferred to the next closest base station using a technique called a **hand-off**: the connection between the cellular telephone and the cellular telephone network is dropped for a fraction of a second, the cellular telephone moves between base stations, and the next base station reestablishes the signal. The area covered by a base station is called a cell.
- **Analog cellular telephone networks** lose data during transmissions when a hand-off occurs, which is unacceptable for data communications.
- **Digital cellular telephone networks** also lose connection during hand-off, but a digital cellular telephone network uses software to recover lost data by requesting that the transceiver resend the data.
- Digital cellular telephone networks trap and correct errors. Analog cellular telephone networks lack error-control capability.
- Analog networks transmit one long burst of information over a communications channel that can either be used for transmitting or receiving information but not both simultaneously, which is called half-duplex.
- In contrast, digital cellular telephone networks transmit information in small packets, called frames or cells.

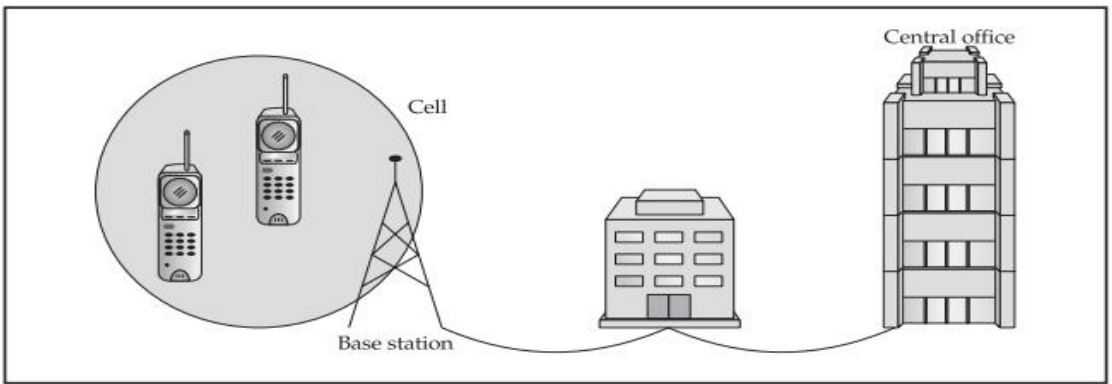



**Figure 2-10.** *Fixed transceivers called base stations form a communications cell.*

## Cellular Digital Packet Data:

- IBM pioneered digital cellular telephone networks with the introduction of their Cellular Digital Packet Data (CDPD) protocol, commonly known as IP wireless.
- IP wireless requires that an Internet protocol (IP) address be assigned to each cellular transceiver in the cellular telephone network.
- An IP address uniquely identifies each device within the cellular telephone network and is used to reestablish a connection if communication is lost during a hand-off.
- Base stations have multiple entry points called ports, each of which is identified by a unique port number.
- A **transceiver** is assigned to a base station port in the cellular telephone network. A transceiver continues to transmit to the port number until a handoff occurs, at which time the transceiver is assigned another port number associated with the next base station.
- IBM developed a special modem called a CDPD modem for transmitting digital information over an analog cellular telephone network.
- The **CDPD** modem transmits small bursts of encrypted data, which frees the communication channel between bursts to transmit error messages.
- Speed is the major stumbling block in using a cellular telephone network to transmit data.
- The standard analog transmission rate of a cellular telephone network is 9,600 bits per second, which is increased to 14,400 bits per second using CDPD. These speeds are sufficient to transmit delivery information, inquire about the status of an order, or provide remote access to email, but are insufficient for full Internet access.

## Digital Wireless Transmissions:

- A digital cellular telephone network can transmit both voice and data simultaneously using multiplex transmission.

- There are **three multiplex transmission methods** used on a **digital cellular telephone network**: **Code Division Multiple Access (CDMA), Time Division Multiple Access (TDMA), and a third-generation wireless standard called 3G.**
- The cellular telephone temporarily uses on-board memory in transceivers to store data to keep transmissions flowing during a hand-off. This is called a **soft hand-off**.
- The 3G multiplexing technique uses either CDMA or TDMA to increase the throughput to 56 kilobits per second.

## Cell Phones and Text Input:

- Traditional cellular telephones have a keypad that contains numbers and letters.
- Today customers expect to be able to enter textual information using the cellular telephone keypad. However, there are two problems with the keypad. First, the keypad doesn't contain the letters Q or Z.
- And each numeric key, except for the first key, contains three letters. A common solution to this problem is for software in the cellular telephone to count the number of times a key on the keypad is pressed to determine which letter of the alphabet was entered. Another solution is to use T9 technology.
- T9 technology uses special glasses that track eye movement, enabling a person to type by moving her eyes in one of eight directions.
- Multiple letters are assigned to each direction. An algorithm was developed that predicted which one of the multiple letters a person wanted to type based on the previous letters that she selected.

## Messaging:

- wireless mobile communications devices offer text messaging services that enable short textual messages to be sent to the device from any device that has access to the service.
- Cellular telephone companies offer **three types of messaging services**: **Short Message Service (SMS), Cell Broadcast Service (CBS), and Unstructured Supplementary Services Data (USSD).**
- **SMS type** of messaging is capable of sending a maximum of 160 characters on the control channel of a cellular telephone network. A control channel is a communications channel used to manage cellular telephone calls. SMS messaging uses store-forwarding technology, where the message is temporarily stored in a mailbox before being delivered to the receiver.
- **The CBS type** of messaging broadcasts a maximum of 15 pages of 93 characters per page to every device on the network. Everyone on the network receives the same message, which is why CBS messaging has had limited success in the market.
- **The USSD type** of messaging transmits a maximum of 182 characters along the control channel, similar to SMS messaging. However, USSD messaging does not use

storeforwarding technology. Instead, USSD messaging sends the message directly to the receiver, which enables the receiver to respond instantaneously.

## Personal Digital Assistants:

- A personal digital assistant (PDA) is probably the most commonly used small mobile computing device next to a cellular telephone. PDAs are lightweight and small enough to fit in a pocket, and they run essential applications.
- There are **three commonly used operating systems** on a **PDA**: **EPOC**, **Palm OS**, and **Windows CE**.
- **EPOC** is used in the Psion product line,**Palm OS** in the Palm PDAs, and **Windows CE** on various pocket PC devices.
- Memory is precious in a PDA. A PDA does not have permanent storage, therefore all the applications and data running in a PDA must reside in memory.
- PDAs use ROM and RAM. ROM is used to store bundled applications from the factory.
- These include a word processor, spreadsheet, diary, telephone directory, and other kinds of programs.
- PDAs use **one of three types of RAM**: **Dynamic RAM (DRAM), Enhanced Data Output (EDO), and Synchronous Dynamic RAM (SDRAM).**
- **DRAM** is the least expensive RAM. **EDO** is found in some PDAs, and **SDRAM** is very rarely used.
- The amount of RAM that affects performance depends on the PDA's operating system. Windows CE requires more memory (32MB) to perform basic functions than a Palm (4MB).
- Some PDAs have an expansion slot for Compact Flash (CF+) cards that contain components such as a modem, cellular telephone, network card used to connect to a local area network, or additional memory that slips into an expansion slot on the PDA to enhance the PDA's functionality.

# J2ME Architecture and Development Environment

## J2ME Architecture:

- The modular design of the J2ME architecture enables an application to be scaled based on constraints of a small computing device. J2ME architecture consists of layers located above the native operating system, collectively referred to as the Connected Limited Device Configuration (CLDC). The CLDC, which is installed on top of the operating system, forms the run-time environment for small computing devices. The J2ME architecture comprises three software layers (Figure 3-1). 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.

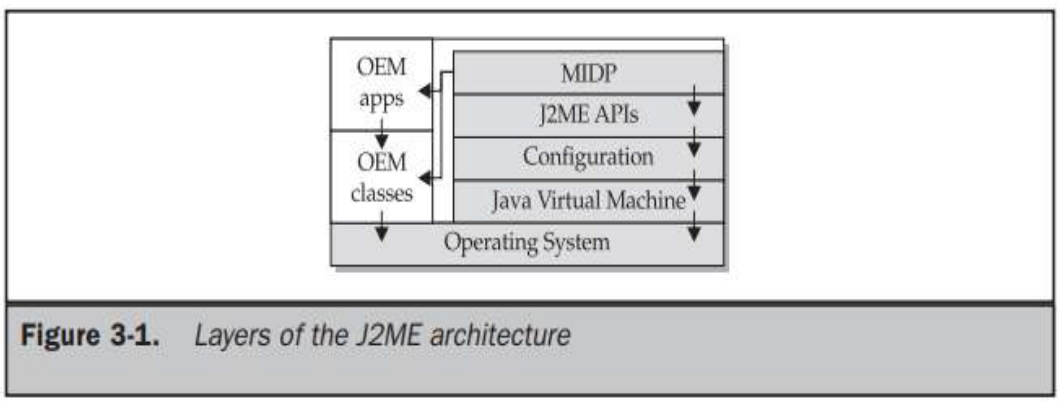



**Figure 3-1.** *Layers of the J2ME architecture*

- 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.

## Small Computing Device Requirements:

- There are minimum resource requirements for a small computing device to run a J2ME application. First 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. 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. 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 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.

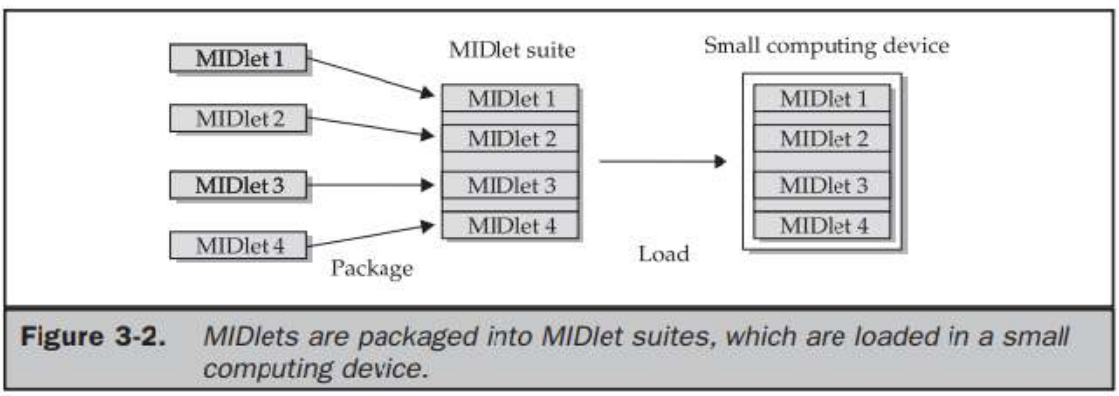



**Figure 3-2.** *MIDlets are packaged into MIDlet suites, which are loaded in a small computing device.*

- 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. A MIDlet suite is installed, executed, and removed by the application manager running on the device. The application manager also makes the Java archive (JAR) file and the Java application descriptor (JAD) file available to members of the MIDlet suite.

**Inside the Java Archive File:**

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

**Table 3-1.** *Attributes of the Manifest File*

- The manifest file's extension is changed to .mf when the MIDlet is prepared for deployment.
  **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 MIDlet-*n* attribute can contain three values that describe the MIDlet. A comma Separates each value. The first value is the name of the MIDlet, which is BestMIDlet. Next is an optional value that specifies the icon that will be used with the MIDlet. In this example, BestMIDlet.png is the icon. The icon must be in the PNG image format. And the last value for the MIDlet-*n* attribute is the MIDlet class name, which is Best.BestMIDlet. The application manager uses the class name to load the MIDlet.

## Inside the Java Application Descriptor 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.
  AJAD 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

| 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). |

**Table 3-2.** *Attributes for a JAD File*

- A developer can include application attributes in a JAD file. An application attribute is a name:value pair that contains a value unique to the application.

## MIDlet Programming:

- Programming a MIDlet is similar to creating a J2SE application in that you define a class and related methods. 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 (Figure 3-3). 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.
- public class BasicMIDletShell extends MIDlet { public void startApp() { } public void pauseApp() { } public void destroyApp( boolean unconditional) { } }

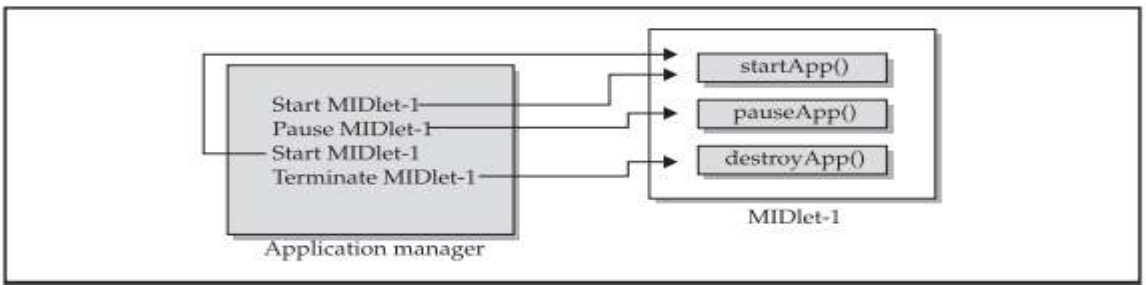



**Figure 3-3.** *The application manager calls methods of a MIDlet.*

- 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. 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 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. 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. A CommandListener monitors user events with a screen and causes the appropriate code to execute based on the current event.

**User Interfaces:**

- The design of a user interface for a MIDlet depends on the restrictions of a small computing device. 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 user interface elements. Text is placed on a form as a StringItem, a List, a ChoiceGroup, and a Ticker. 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. And a 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). 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:

- CDC implements the full J2SE available, but CLDC implements a stripped-down J2SE because of the limited resources in small computing devices. Floating-point math is probably the most notable missing feature of J2ME. Floatingpoint math requires special processing hardware to perform floating-point 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. 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. 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. 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. , 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.

## J2ME Software Development Kits:

- A MIDlet 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/).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. It is best to accept the default directory, although you are free to choose a different directory for the Java development kit. 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 (see "Setting the Path in Windows" sidebar). This enables you to invoke the Java compiler from anywhere on your computer.

**Setting the Path in Windows**

**Windows 2000 and Windows NT**

1. Choose System from the Control Panel.
2. Select Environment or Advanced/Environment.
3. Locate the PATH environment variable.
4. Enter the directory at the end of the path. Be sure to separate entries with a semicolon.

**Windows 98 and Windows 95**

1. Select Start.
2. Select Run.
3. Enter **sysedit**.
4. Select OK.
5. Locate the autoexec.bat dialog box.
6. Add the directory to 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. 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". 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. Some versions create a midp-fcs directory, while the 1.0.3 version creates a %J2ME_HOME%\ midp1.0.3fcs directory. 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;. Set the MIDP_HOME environment variable to d:\j2me\midp1.0.3fcs.

**Hello World J2ME Style:**

- create the first MIDlet once the Java development kit, Connected Limited Device Configuration (CLDC), and Mobile Information Device Profile (MIDP) are installed.
- keeping tradition alive, let's begin by creating a directory structure within which you can create and run MIDlets.
- Here are the directories that are used for examples in this chapter:
  - j2me
  - j2me\src
  - j2me\src\greeting
  - j2me\tmp_classes
  - j2me\midlets.
- The HelloWorld MIDlet shows how to create a simple MIDlet that can be invoked directly from the class and from a Java archive file
- Let's begin by creating the HelloWorld MIDlet. Enter the code shown in Listing 3-4 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 that are found in nearly all MIDlets. These are to display a text box and a command on the screen, then 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(), which are discussed earlier in this chapter. 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.
- import javax.microedition.midlet.*; import javax.microedition.lcdui.*; public class HelloWorld extends MIDlet implements CommandListener { private Display display ;

private TextBox textBox ; private Command quitCommand; public void startApp() { display = Display.getDisplay(this); quitCommand = new Command("Quit", Command.SCREEN, 1); textBox = new TextBox("Hello World", "My first MIDlet", 40, 0); textBox .addCommand(quitCommand); textBox .setCommandListener(this); display .setCurrent(textBox ); } public void pauseApp() { } public void destroyApp(boolean unconditional) { } public void commandAction(Command choice, Displayable displayable) { if (choice == quitCommand) { destroyApp(false); notifyDestroyed(); } } }

- 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. Prior to invoking the notifyDestroyed() method, a MIDlet should have completed its own garbage collection.

**Compiling Hello World:**

- The Hello World source code files should be saved in the new j2me\src\greeting directory as HelloWorld.java. 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, as described previously in this chapter. The preverification generates a modified class file. Make j2me\src\greeting the current directory, and then enter the following command at the command line. javac -d d:\j2me\tmp_classes -target 1.1 -bootclasspath d:\j2me\midp1.0.3fcs\classes HelloWorld.java.

**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. Click the right telephone handset icon to close the MIDlet. 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.
- MIDlet-1: HelloWorld, , greeting.HelloWorld MIDlet-Name: Hello World MIDlet-Version: 1.0 MIDlet-Vendor: Jim MIDlet-1: HelloWorld, /greeting/myLogo.png, greeting.HelloWorld MicroEdition-Configuration: CLDC-1.0 MicroEdition-Profile: MIDP-1.0.

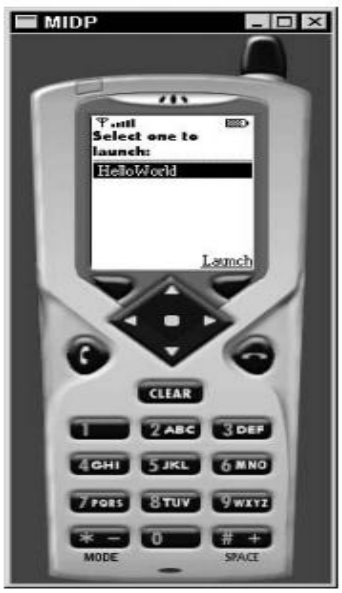



**Figure 3-4.** *The HelloWorld MIDlet running in the emulator*

- 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. save the JAD file in the j2me/src/greeting directory. MIDlet-Name: Hello World MIDlet-Version: 1.0 MIDlet-Vendor: Jim, 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. Click the right telephone handset icon to close the MIDlet. midp -classpath HelloWorld.jar -Xdescriptor HelloWorld.jad.

**What to Do When Your MIDlet Doesn't Work Properly:**

- Sometimes a MIDlet won't compile or run properly. 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. This results in not enough room to store the complete value of an environment variable such as the PATH. Many types of errors can occur during the compiling and packaging process. Some are syntax errors, which you'll be able to fix quickly by reviewing the source code. 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.

**Multiple MIDlets in a MIDlet Suite:**

- In the real world, multiple MIDlets are distributed in a single MIDlet suite. The application manager then displays each MIDlet as a menu option, enabling the user to run one of the MIDlets.

## J2ME Wireless Toolkit:

- Go through the textbook **Pg.No:60-66**

# UNIT II

## J2ME Best Practices and Patterns:

### The Reality of Working in a J2ME World:

- A small computing device has a radically different hardware configuration than traditional computing devices such as desktop computers and servers. It is for this reason that you must take into account the device's hardware configuration when designing your J2ME application.
- **differences between traditional computing devices and small computing devices**

- Traditional computing devices are under continuous power from the power grid, while some small computing devices such as cellular telephones rely on battery power that diminishes during the course of operation. A power grid powers other small computing devices such as set-top boxes and appliances. Another important difference between traditional computing devices and small computing devices is the network connection. Unlike traditional computing devices, mobile small computing devices connect to a network via a radio or infrared connection whose quality varies depending on the distance of the device from a network receiver and the strength of the signal generated by the device. Some nonmobile small computing devices such as set-top boxes use a hard-wired network connection similar to traditional computing devices. Inconsistency in a network connection and the diminishing longevity of power typically require the user of a small computing device to synchronize data and applications frequently with a desktop computer or server. As you learned in Chapter 1, programs and data are stored in a small computer device's memory, commonly referred to as primary storage. These are lost when the device drops power, although many devices have a secondary battery to retain programs and data as long as possible. Once lost, programs and data must be reloaded into the device. Secondary storage is not usually available on a small computing device. Therefore, a J2ME application should rely on data stored offline in a desktop computer or server rather than data stored in the device's primary storage. Data stored offline can be reloaded into the device using a network connection. Don't expect a mobile small computing device to transmit and receive data at the same rate as a device on a hard-wired network. 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. A bandwidth is the number of communications channels available to transmit bits of data simultaneously.

**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.
- **Keep Applications Simple:**
    1. u must adapt to a new mind-set when creating applications for small computing devices because of limited resources available and the inability to easily expand resources to meet application requirements
- **Keep Applications Small:**
    2. The size of your J2ME application is critical to deploying the application efficiently. The best practice is to remove unnecessary components of your application in order to reduce the size of the overall application.
- **Limit the Use of Memory:**
    3. There are two types of memory management that should be used in the 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. A boolean value requires less memory and therefore should be used in place of an int. This and similar data management subtleties usually have little or no noticeable impact on a non-J2ME application. Peak time memory management requires you to manage garbage collection. J2ME does have a garbage collector, but as with J2SE, you don't know when the garbage collector will collect your garbage.
- **Off-Load Computations to the Server:**
    4. 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. The alternative is to build a client-service J2ME application or web services J2ME application. There are two levels of operation in a client-service application. These are the client level and the server level. The small computing device runs the client level that provides user interface and presentation functionality to the application. The server-side level processes client requests and returns the result to the small computing device for presentation to the user. There are three tiers in web services. The first layer is the client tier, sometimes referred to as the presentation tier. This is where a person interacts with an application. The second layer contains the business logic that is used to fulfill requests from a client by calling appropriate software on the processing tier. Processing software returns results to the business logic layer, and in turn, those results are returned to the client for presentation to the user.

- **Manage Your Application's Use of a Network Connection:**
    5. Besides lightening the processing load on the small computing device, you must also be concerned about the availability of a network connection. Some 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 receiverCellular telephone networks use technology that attempts to maintain connection as the mobile device moves from one cell to another cell.
- **Simplify the User Interface:**
    6. Most desktop applications have a standard set of graphical user interface objects such as text boxes, combo boxes, radio buttons, check boxes, and push buttons. There is a standard display and input for desktop computers, but you cannot say the same about small computing devices. The variety of shapes and hardware configurations found in devices classified as small computing devices makes it nearly impossible to standardize on a set of user interface objects for these devices.
- **Use Local Variables:**
    7. Limited resource is the theme that echoes through design considerations for applications that run on small computing devices. Failure to seriously recognize this theme will result in your application being unable to run on many small computing devices. Therefore, it is critical to evaluate processing requirements of each routine within your application. Data storage is a key area within an application for reducing excessive processing. In many applications, developers assign values to data members of a class rather than using a local variable. Assigning data to a class member adheres to object-oriented design philosophy, which is prevalent in application design.
- **Don't Concatenate Strings:**
    8. Concatenating strings is another processing drain that can be avoided by designing an application to eliminate concatenations or at least reduce the number of concatenations to the minimum necessary to achieve the objective of the application. A string is an array of characters terminated by a NULL and stored sequentially 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.

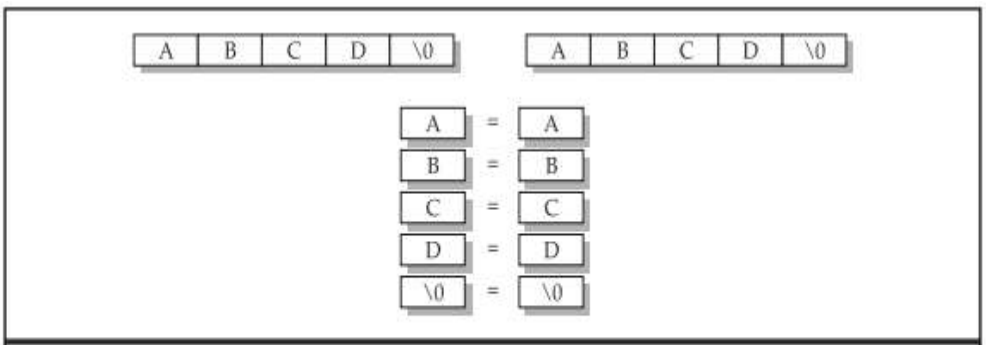



**Figure 4-5.** *Ten reading instructions and five comparison instructions are necessary to compare two strings.*

- **Avoid Synchronization:**
  - **9.** It is very common for developers to invoke one or multiple threads within an operation. Invoking a thread is a way of sharing a routine among other operations. Each operation invokes the sort routine independent of other operations, although the same code is being executed for all operations. Deadlocks and other conflicts might arise when multiple operations use the same routine.

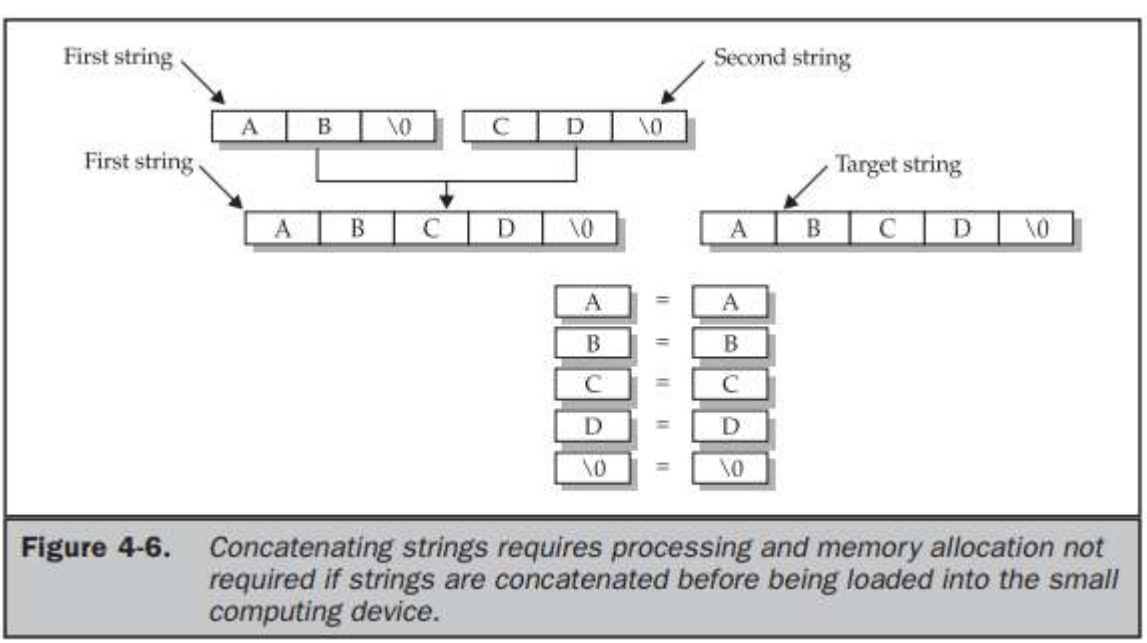



**Figure 4-6.** *Concatenating strings requires processing and memory allocation not required if strings are concatenated before being loaded into the small computing device.*

Another way to increase performance is to avoid using synchronization where possible. Synchronization requires additional processing steps that are not necessary when synchronization is deactivated.

- **Thread Group Class Workaround:**
  - **10.** 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. Grouping thread objects is made possible by the ThreadGroup class, but J2ME does not support this class.

- **Upload Code from the Web Server:**
  **11.** 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.
- **Reading Settings from JAD Files:**
  **12.** 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
- **Populating Drop-down Boxes:**
  **13.** A drop-down box is a convenient way for users to choose an item from a list of possible items, such as an abbreviation for a state. 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.
- **Minimize Network Traffic:**
  **14.** 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 the information from the user that is required by the process at one time, and then forward the information to the server when invoking the process.
- **Dealing with Time:**
  **15.** Desktop computers and servers are stationary, and therefore current time reflects the time zone where these devices are located. The problem of time-sensitive data is further compounded by the fact that the date/ time setting is device dependent. The best practice is to always store time based on Greenwich Mean Time (GMT) by using the getTime() method of the Date class.
- **Automatic Data Synchronization:**
  **16.** 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. Data is permanently stored in secondary storage on a traditional computing device such as a desktop computer or server. A good practice is to build into your J2ME application a routine that automatically uploads the latest data when the J2ME application is invoked.
- **Updating Data that Has Changed:**
  **17.** 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. A good practice is to offer the user of your application 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. And

only the changed data is 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.

- **Be Careful of the Content of the startApp() Method:**

**18.** Tips for Developing J2ME Applications

■ Applications are typically single-threaded.

■ One application runs at a time.

■ Applications are event driven.

■ Users change from one application to another rather than terminating an application. ■ Mobile small computing devices are used intermittently.

■ Applications use multiple subscreens, each displaying only relevant information.

■ Mobile small computing devices are typically used in two-minute sessions 30 times a day.

■ Applications must accomplish a task within two minutes; otherwise the user is likely to turn off the mobile small computing device.

■ Limit user input to a few keystrokes. Develop a PC-based component of your application that is used for data input.

■ Users want an instant response from an application.

■ Off-load processing to a server or desktop computer.

■ Avoid power-consuming tasks such as communications, animation, and sound.

■ Reduce data communication to the bare minimum because users pay for transmission by byte, usually in the range of 50,000 to 300,000 bytes per month.

■ Preload as many files as possible into a mobile small computing device in order to reduce data transmissions.

**J2ME User Interfaces:**

- A user interface is a set of routines that displays information on the screen, prompts theuser to perform a task, and then processes the task.
- The device's application manager passes the selection to the application,where it is compared with known options. If a match occurs, the application performs the steps necessary to process the option.
- **A developer can use one of three kinds of user interfaces for an application.**
  a) command, form, or canvas. 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.
  b) 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.

c) 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:**

- The device's screen is referred to as the display, and you interact with the display by obtaining a reference to an instance of the MIDlet'sDisplay class.
- Each MIDlet has one and only one instance of the Display class. 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 has two subclasses.
  a) Screen class and
  b) the Canvas 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 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.
  **private Display display;**
  **display = Display.getDisplay(this);**

**Command Class:**

- create an instance of the Command class by using the Command class constructor within your J2ME application.
- The Command class constructor requires three parameters.
  a) command label,
  b) the command type, and
  c) the command priority.
- The Command class constructor returns an instance of the Command class.
- 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.
- The **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.
  **cancel = new Command("Cancel", Command.CANCEL, 1);**

| 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 |

**Table 5-1.** *Command Types*

- 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 your application for processing. Priority indicates your preference as to the importance of each command object created by your application. Priority is established by the value that you 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.

**CommandListener:**

- 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,

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

**Example:**

```
import javax.microedition.midlet.*;
```

```
import javax.microedition.lcdui.*;
public class OnlineHelp extends MIDlet implements CommandListener
{
private Display display;
private Command back;
private Command exit;
private Command help;
private Form form;
private TextBox helpMesg;
public OnlineHelp()
{
display = Display.getDisplay(this);
back = new Command("Back", Command.BACK, 2);
exit = new Command("Exit", Command.EXIT, 1);
help = new Command("Help", Command.HELP, 3);
form = new Form("Online Help Example");
helpMesg = new TextBox("Online Help", "Press Back to return
to the previous screen or press Exit to close this
program.", 81, 0);
helpMesg.addCommand(back);
form.addCommand(exit);
form.addCommand(help);
form.setCommandListener(this);
helpMesg.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 == back)
{
display.setCurrent(form);
}
else if (command == exit)
{
destroyApp(false);
notifyDestroyed();
```

```
}
else if (command == help)
{
display.setCurrent(helpMesg);
}
}
}
```


**Figure 5-3.** *Select the button below the Help command to display the help screen on the default color phone emulator.*


**Figure 5-4.** *Select the button below the Back command to return to the previous screen in the default color phone emulator.*

## 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. 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. 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.
- 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.
- The itemStateChanged() method processes the change before focus is set on the other instance.

**Item Listener:**

- 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.

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

- Since 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.

**Example:**

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class RadioButtons extends MIDlet
implements ItemStateListener, CommandListener
{
private Display display;
private Form form;
private Command exit;
private Item selection;
private ChoiceGroup radioButtons;
private int defaultIndex;
private int radioButtonsIndex;
public RadioButtons()
{
display = Display.getDisplay(this);
radioButtons = new ChoiceGroup(
"Select Your Color",
Choice.EXCLUSIVE);
radioButtons.append("Red", null);
radioButtons.append("White", null);
radioButtons.append("Blue", null);
radioButtons.append("Green", null);
defaultIndex = radioButtons.append("All", null);
radioButtons.setSelectedIndex(defaultIndex, true);
exit = new Command("Exit", Command.EXIT, 1);
form = new Form("");
radioButtonsIndex = form.append(radioButtons);
form.addCommand(exit);
form.setCommandListener(this);
form.setItemStateListener(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();
}
}
public void itemStateChanged(Item item)
{
if (item == radioButtons)
{
StringItem msg = new StringItem(Your color is ",
radioButtons.getString(radioButtons.getSelectedIndex()));
form.append(msg);
}
}
}
```

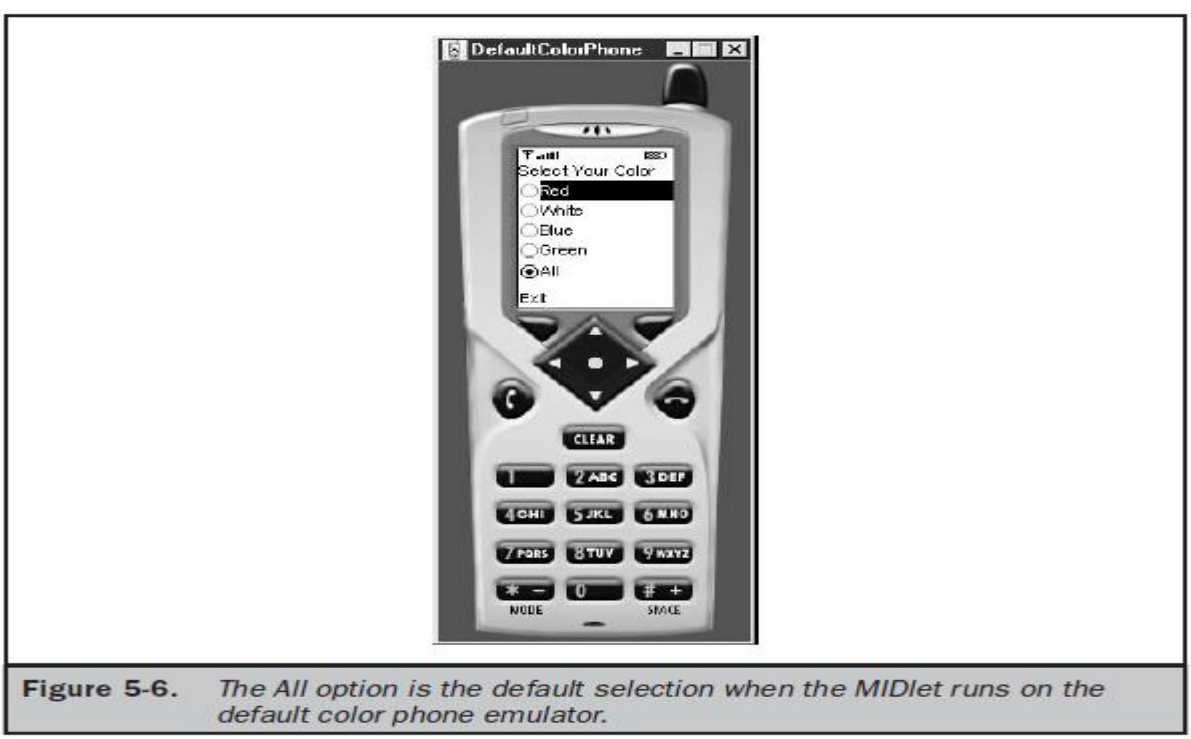



**Figure 5-6.** *The All option is the default selection when the MIDlet runs on the default color phone emulator.*

## 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. 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. can regain a little control of the MIDlet's operation by causing a MIDletStateChangeException to be thrown.
- A MIDletStateChangeException is usedto temporarily reject a request from the application manager either to start the MIDlet (startApp()) or to destroy the MIDlet (destroyApp()).
- AMIDletStateChangeException cannot be thrown within the pauseApp() method.

**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;
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 (exitFlag == false)
{
StringItem msg = new StringItem ("Busy", "Please try again.");
form.append(msg);
destroyApp(false);
}
else
{
destroyApp(true);
notifyDestroyed();
}
}
catch (MIDletStateChangeException exception)
{
isSafeToQuit = true;
}
}
}
}
```

- 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.
- The MIDlet successfully terminates the second time the user selects the Exit button.


**Figure 5-8.** *The Exit command must be selected twice to terminate the MIDlet on the default color phone emulator.*

- 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 MIDletshould not be terminated.
- Likewise in the constructor there are statements that obtain instances to the Display class, Command class, and Form class, each of which is assigned to the proper reference.
- The command is also associated with the form using the addCommand() method, and a CommandListener is associated with the form.

**Example:**

```
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;
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 (exitFlag == false)
{
StringItem msg = new StringItem (
"Busy", "Please try again.");
form.append(msg);
destroyApp(false);
}
else
{
destroyApp(true);
notifyDestroyed();
}
}
catch (MIDletStateChangeException exception)
{
isSafeToQuit = true;
}
}
}
}
```


**Figure 5-8.** *The Exit command must be selected twice to terminate the MIDlet on the default color phone emulator.*

## The Palm OS Emulator:

- TheROM file contains the Palm OS required for the emulator to properly perform like a Palm PDA.
- 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.
- Several ROM files are available for download, each representing a different version of the Palm OS and each suited for a particular Palm product.
- 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 MIDlet is Palm device specific, you'll need todownload the ROM file that corresponds to the Palm OS that runs on that Palm device.
- 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.
- be prompted to enter the location of the ROM file on your hard disk into a dialog box the first time that you run the Palm OS emulator.
- Subsequently, the Palm OS emulator uses that ROM file.

# UNIT-III

## High-Level Display:

- The display is a crucial component of every J2ME application since it contains objects used to present information to the person using the application and in many cases prompts the person to enter information that is processed by the application.
- The J2ME Display class is the parent of Displayable, **The Displayable class has two subclasses of its own: Screen and Canvas.**
- The Screen class is used to create high-level J2ME displays in which the methods of its subclasses handle details of drawing objects such as radio buttons and check boxes.
- In contrast, the Canvas class and its subclasses are used to create low-level J2ME displays.

## Screen Class:

- Screen class contain methods that generate radio buttons, check boxes, lists, and other familiar objects that users expect to find on the screen when interacting with your application.
- Display class hierarchy, which helps you learn the inheritance structure of the Screen class.
- A Displayable object is any object that can be displayed on the small computing device's screen.

```
public class Display
    public abstract class Displayable
        public abstract class Screen extends Displayable
            public class Alert extends Screen
            public class Form extends Screen
            public class List extends Screen implements Choice
                public abstract class Item
                    public class ChoiceGroup extends Item implements Choice
                    public class DateField extends Item
                    public class TextField extends Item
                    public class Gauge extends Item
                    public class ImageItem extends Item
                    public class StringItem extends Item
                public class Command
                public class Ticker
                public class Graphics
                public interface Choice
        public abstract class Canvas extends Displayable
            public class Graphics
```

- **The Screen class has its own set of derived classes. These are**
    a) TextBox,
    b) List,

c) Alert,
d) Form, and
e) Item classes.

- **The Canvas class also has its own derived class**
  a) Graphics class,
- The **TextBox class** is used to display multi-line text on the screen.
- The **List class** is used to display a list of items, as in a menu, and enables the user to choose one of those items.
- The **Alert class** displays a dialog box containing a message such as a warning.
- And the Form class is a container class that can display multiple classes derived from the Item class.
- **The Item class has six derived classes**, any number of which can be displayed within a Form object on the screen:
  - **ChoiceGroup** class used to display radio buttons and check boxes
  - **DateField** class used for inputting a date into an application
  - **TextField** class used for inputting text into an application
  - **Gauge** class used to graphically show progress
  - **ImageItem** class used to display an image stored in a file
  - **StringItem** class used to display text on the screen
- The **Command class** is used to create a Command object that can be associated with practically any class except the Alert class.
- The **Ticker** is a variable of the Screen class that causes text to scroll on the screen like a stock exchange ticker tape.
- The **Graphics class** is a base class used by derived classes to create and display custom graphical images on the screen. Objects that display options to the person using an application implement the Choice interface.

## Alert Class:

- An alert is a dialog box displayed by your program to warn a user of a potential error such as a break in communication with a remote computer.
- An alert can also be used to display any kind of message on the screen, even if the message is not related to an error.
- An alert is an ideal way of displaying a reminder on the screen.
- implement an alert by creating an instance of the Alert class in your program using the following statement.
- Once created, the instance is passed to the setCurrent() method of the Display object to display the alert dialog box on the screen.

```
alert = new Alert("Failure", "Lost communication link!", null, null);
display.setCurrent(alert);
```

- The Alert constructor requires **four parameters**.
  1. The first parameter is the title of the dialog box, which is “Failure” in this example.
  2. The next parameter is the text of the message displayed within the dialog box. “Lost communication link!” is the text that appears when the Failure dialog box is shown on the screen.

3. The third parameter is the image that appears within the dialog box. The previous example doesn't use an image; therefore the third parameter is set to null.
4. The last parameter is the AlertType.

- The AlertType is a predefined type of alert. A word of caution: An alert dialog box is not designed to retrieve input from user other than the selection of the OK button to close the dialog box. This means displayable objects such as **ChoiceGroup and TextBox** cannot be used within an alert dialog box. We cannot insert your own Command objects as buttons.
- An alert dialog box reacts in one of two ways depending on the value of the default timeout for the Alert object.
- An alert dialog box is referred to as a ***modal* dialog box** if the user must select the OK button to terminate the dialog box. Otherwise, it is considered a ***timed* dialog box** that terminates when the default timeout value is reached.

| Type | Description |
|---|---|
| ALARM | Your request has been received. |
| CONFIRMATION | An event or processing is completed. |
| ERROR | An error is detected. |
| INFO | A nonerror alert occurred. |
| WARNING | A potential error could occur. |

**Table 6-1.** *Predefined AlertTypes for the Alert Object*

## setTimeout():

- The value passed to the setTimeout() method determines whether an alert dialog box is a modal dialog box or a timed dialog box.
- The setTimeout() method has **one parameter**,which is the default timeout value.
- Use Alert.FOREVER as the default timeout value for amodal alert dialog box, or pass a time value in milliseconds indicating time to terminate thealert dialog box.
- **The following example illustrates how to create a modal alert dialog box:**

```
alert = new Alert("Failure", "Lost communication link!", null, null);
alert.setTimeout(Alert.FOREVER);
display.setCurrent(alert);
```

## getDefaultTimeout():

- we can always retrieve the current default timeout by calling the getDefaultTimeout() method of the instance of the Alert class.
- The getDefaultTimeout() method returns theinteger value of Alert.FOREVER or the default timeout in milliseconds.

- The device's application manager determines the screen that appears when the user dismisses the alert dialog box.
- can control what appears following the dialog box by passing reference to the next object as the second parameter to the setCurrent() method.
- The **second parameter** is reference to the displayable object that appears on the screen once the alert dialog box is closed, Once the user selects OK to dismiss the alert dialog box, the device's application manager displays the instances of the Form object, enabling the user to reestablish communication with the remote computer.

```
form = new Form("Communication Link");
alert = new Alert("Failure", "Lost communication link!", null, null);
alert.setTimeout(Alert.FOREVER);
display.setCurrent(alert, form);
```

- An alternative way to control which instance of a class appears on the screen oncean alert dialog box is terminated is simply to invoke the setCurrent() method twice.
- Pass reference to the instance of the Alert class to the first invocation of the setCurrent(), and then pass reference to the next instance the next time that the setCurrent() method is called, as shown below.
- **Calling the setCurrent() method twice using one parameter or once using two parameters achieves the same results without penalties.**

```
form = new Form("Communication Link");
alert = new Alert("Failure", "Lost communication link!", null, null);
alert.setTimeout(Alert.FOREVER);
display.setCurrent(alert);
display.setCurrent(form);
```

**Example:**

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class DisplayAlert extends MIDlet implements CommandListener
{
private Display display;
private Alert alert;
private Form form;
private Command exit;
private boolean exitFlag;
public DisplayAlert()
{
exitFlag = 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 (exitFlag == false)
{
alert = new Alert("Busy", "Please try again.",
null, AlertType.WARNING);
alert.setTimeout(Alert.FOREVER);
display.setCurrent(alert, form);
destroyApp(false);
}
else
{
destroyApp(true);
notifyDestroyed();
}
}
catch (MIDletStateChangeException exception)
{
exitFlag = true;
}
}
}
}
```


**Figure 6-1.** *An alert dialog box displays a message on the screen.*

**Alert Sound:**

- Each AlertType has an associated sound that automatically plays whenever the alert dialog box appears on the screen.
- The sound, which is different for each **AlertType**, is used as an audio cue to indicate that an event is about to occur. An audio cue can be sounded without having to display the alert dialog box.

**playSound():**

- We can do this by calling the **playSound()** method and passing it reference to the instance of the Display class.
- The sound associated with the **AlertType WARNING** is heard when the **playSound()** method is called.

```
if (exitFlag == false)
{
        AlertType.WARNING.playSound(display);
        destroyApp(false);
}
```

## Form Class:

- The Form class is a container for other displayable objects that appear on the screen simultaneously.

- Any derived class of the Item class can be placed within an instanceof the Form class.
- Small computing device screens vary in size, so you can expect that some instances within the instance of the Form class won't fit on the screen. However, devices typicallyimplement scrolling, which allows the user to bring instances out of view onto the screen.
- **An instance is placed with the instance of the Form class by calling one of two methods.**
- These are **insert() method and append() method**.
- The **insert() method** places the instance in a particular position on the form as specified by parameters passed to the insert() method.
- The **append() method** places the instance after the last object on the form. The append() method is called once when both instances are created. Reference to the StringItem instance is then passed to the form, thereby placing the StringItem instance as the last object on the form.

```
private Form form;
private StringItem message;
form = new Form("My Form");
message = new StringItem("Welcome, ", "glad you could come.");
form.append(message);
```

- Each instance placed on a form has an index number associated with it, beginningwith the value zero.
- we can use the index number to reference the instance within the MIDlet, for example, when you want to insert an instance onto the form.
- The following segment of code shows you how to insert another StringItem instance onto a form before the first StringItem instance.
- An int is also declared and is used to store the index number of the first StringItem instance placed on the form.
- The same Form instance and StringItem instance as in the previous example are createdand assigned to the proper reference. However, a second StringItem instance is also created.
- Notice that the index number of the first message appended to the Form instance is stored in the index1 variable.
- The **index1 variable is passed as the first parameter to the insert() method** to place the second message on the form before the first message.
- **Reference to the second message is passed as the second parameter to the insert() method.**

```
private Form form;
private StringItem message1, message2;
private int index1;
form = new Form("My Form");
message1 = new StringItem("Welcome, ", "glad you could come.");
message2 = new StringItem("Hello, ", "Mary.");
index1 = form.append(message1);
form.insert(index1,message2);
```

- An alternative to using the insert() and append() methods for associating instances of the Item class with a form is to create an array of instances of the Item class and then pass the array to the constructor when the instance of the Form class is created.
- This is an excellent technique for initially populating the instance of the Form class. You can then use the **insert() method, append() method, set() method, and delete() method** to manage instances of the Item object on the form throughout the life of the MIDlet.

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class CreatingFormWithItems extends MIDlet implements CommandListener
{
private Display display;
private Form form;
private Command exit;
public CreatingFormWithItems ()
{
display = Display.getDisplay(this);
exit = new Command("Exit", Command.SCREEN, 1);
StringItem messages[] = new StringItem[2];
message[0] = new StringItem("Welcome, ", "glad you could come.");
message[1] = new StringItem("Hello, ", "Mary.");
form = new Form("Display Form with Items", messages);
form.addCommand(exit);
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();
}
}
}
```

**set() method:**

- An instance of the Item class that appears on the form can be replaced by another instance of the Item class by calling the set() method.
- **The set() method requires twoparameters**. The **first parameter** is the index number of the instance of the Item class that is being replaced, and the **other parameter** is reference to the instance of the Item object that is replacing the existing Item class.

**delete() method:**

- We can remove an instance of the Item class from a form by invoking the delete() method.
- The delete() method requires **one parameter**, which is the index number of the instance of the Item class that is being removed from the form.

## Item Class:

- An **Item class** is a base class for a number of derived classes that can be contained within a Form class.
- These derived classes are **ChoiceGroup, DateField, Gauge, ImageItem,StringItem, and TextField.**
- Some classes derived from the Item class, such as **ChoiceGroup, DateField, and TextField**, are used for data entry.
- The ChoiceGroup class is used to create check boxes or radio buttons on a form, and the DateField class and TextField class are used to capture date and free form text from the user of the MIDlet.
- The state of an instance of a class derived from the Item class changes whenever a user enters data into the instance, such as when a check box is selected.
- You can capture this change by associating an **ItemStateListener** with an instance of a class derived from an Item class.
- An **ItemStateListener** monitors events during the life of the MIDlet and traps events that represent changes in the state of any Item class contained in a form on the screen.
- The class implementing the **ItemStateListener interface** (the MIDlet in this case) becomes the registered listener (callback) whose **itemStateChanged()** method is called when an item event occurs.
- The device's application manager detects the event and calls the **itemStateChanged()** method of the MIDlet.
- CommandListener when a command event occurs, except when the commandAction() method is invoked.
- Logic within the **itemStateChanged()** method compares the reference to known items on the form and then initiates processing.
- The nature of this processing is application dependent, but processing is likely to retrieve the value that the user entered into the item.
- The **itemStateChanged()** method is defined outside the constructor and contains logic to evaluate the item passed to the method.
- An if statement is used in this example to determine whether the selected item is the instance of the ChoiceGroup class.

- If so, the item is processed according to the business rules of the application.

```
private Form form;
private ChoiceGroup choiceGroup;
….
choiceGroup = new ChoiceGroup("Pick One", Choice.EXCLUSIVE);
form.append(choiceGroup);
form.setItemStateListener(this);
….
public void itemStateChanged(Item item)
{
if (item == choiceGroup)
{
// do some processing
}
}
```

## Choice Group Class:

- Check boxes and radio buttons used in graphical user interfaces for choosing one or multiple choices from a selection of options.
- Likewise, check boxes and radio buttons are used to display selected options that were previously chosen.
- **Check boxes and radio buttons** are often grouped into sets of options, although there are times when one check box, rather than multiple check boxes, is required by an application.
- **Radio buttons** are almost always displayed in a set of radio buttons. **The primary difference between a set of check boxes and a set of radio buttons, besides their obvious appearance, is the number of check boxes or radio buttons that users can select. Users can choose multiple check boxes within a set of check boxes, while they can choose only one radio button within a set of radio buttons.**
- J2ME classifies check boxes and radio buttons as the ChoiceGroup class.
- An instance of the **Choice Group class can be one of two types**: **exclusive or multiple**.
- An ***exclusive*** instance appears as a set of radio buttons, and a ***multiple*** instance contains one or a set of check boxes.
- You determine the format of an instance of a ChoiceGroup class by passing the ChoiceGroup class constructor a choice type,

| Choice Type | Description |
|---|---|
| EXCLUSIVE | Only one selection available at any time (radio button). |
| MULTIPLE | Zero or more selections available at any time (check box). |
| IMPLICIT | Only one selection at any time. The selection generates a command event automatically. No icon is used (menu list). |

**Table 6-2.** *Choice Types for ChoiceGroup Object and List Object*

**itemStateChanged() method:**

- The itemStateChanged() method determines whether the item selected is an instance of the ChoiceGroup. If so, then either the **getSelectedFlags() method or getSelectedIndex() method** must be called to retrieve the item selected by the user.

**getSelectedFlags():**

- The **getSelectedFlags()** method returns an array that contains the status of the selected flag for each member of the instance of the ChoiceGroup class (each radio button or each check box).
- The MIDlet must step through each element of the array to determine whether the selected flag status is true or false.
- If true, the radio button or check box that corresponds to the index of the array element was selected by the user. If false, the user did not make a selection.

**getSelectedIndex():**

- The **getSelectedIndex()** method returns the index number of the item selected by the user, such as a radio button.
- The index number is typically passed to the getString() method, which returns the text of the selected radio button or check box. Instead of using the ItemStateListener and itemStateChanged() methods, you can place a Command on the screen and implement a CommandListener and define an actionCommand() method.

- the device's application manager notifies the CommandListener when a command is selected, and then the actionCommand() method you define in the MIDlet is called.
- The actionCommand() method then calls either the getSelectedFlags() method or the getSelectedIndex() method to identify the item selected by the user.

**Choice Group Class Example:**

- The **append()** method is called once for each check box.
- The **append() method requires two parameters**. The **first parameter** is the label of the check box, and the **other parameter** is reference to an image that appears along with the label. No images are used in this example, so the second parameter is set to a null value.
- The MIDlet performs some interesting processing when the user selects the Process command.

**size():**

- First, an array of boolean types is created. Notice that the dimension of the array is set by calling the size() method of the instance of the ChoiceGroup.
- The size() method returns the number of check boxes in the set. The size() method is also used to create an array of instances of the StringItem object.
- These instances are used to display the user's selections. Next, the boolean array is passed to the getSelectedFlags() method.

- ThegetSelectedFlags() method populates the boolean array with the state of each checkbox.
- A for loop is then used to step through the boolean array, evaluating the value of each array element.
- The picks array length variable is used instead of the size() method to set the maximum iterations of the loop.
- If the value of the boolean array element is true, the MIDlet calls the getString()method, passing it the index number of the check box.
- Each check box is assigned an index number relative to other check boxes within the set and is used to uniquely identify the check box.

**getString():**

- The **getString()** method returns the label of the check box, which is then passed to the **setText()** method of the next instance of the StringItem class and is later displayed on the screen by appending the string to the form.
- The instance of the ChoiceGroup class and the Process command are both removed from the form by calling the **delete() method and the removeCommand() method**, respectively.

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class CheckBoxes extends MIDlet implements CommandListener
{
private Display display;
private Form form;
private Command exit;
private Command process;
private ChoiceGroup movies;
private int movieIndex;
public CheckBoxes()
{
display = Display.getDisplay(this);
movies = new ChoiceGroup("Select Movies You Like to See",
Choice.MULTIPLE);
movies.append("Action", null);
movies.append("Romance", null);
movies.append("Comedy", null);
movies.append("Horror", null);
exit = new Command("Exit", Command.EXIT, 1);
process = new Command("Process", Command.SCREEN,2);
form = new Form("Movies");
movieIndex = form.append(movies);
form.addCommand(exit);
form.addCommand(process);
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 == process)
{
boolean picks[] = new boolean[movies.size()];
StringItem message[] = new StringItem[movies.size()];
movies.getSelectedFlags(picks);
for (int x = 0; x < picks.length; x++)
{
if (picks[x])
{
message[x] = new StringItem("",movies.getString(x)+"\n");
form.append(message[x]);
}
}
form.delete(movieIndex);
form.removeCommand(process);
}
else if (command == exit)
{
destroyApp(false);
notifyDestroyed();
}
}
}
```

## Example for Radio Buttons:

the type
is EXCLUSIVE, which limits selection to only one set of choices within the group.

set a default
selection by first storing the index number of the Male radio button that is returned by the append() method. This index number is then passed as the first parameter to the setSelectedIndex() method. The setSelectedIndex() method's second parameter is a boolean value indicating whether the radio button is on or off. In this example, a true

value is passed to turn on the radio button. The MIDlet invokes the getSelectedIndex() method of the gender object if the incoming
command is the Process command. The getSelectedIndex() returns an integer representing the index of the gender object selected by the user. The index is passed to the getString() method, which returns the radio button's label and assigns the label to an instance of the StringItem class. This instance is then displayed on the form by calling the append() method; afterward the gender instance and the Process command are removed from the form.

**Example for RadioButton:**

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class RadioButtons extends MIDlet implements CommandListener
{
private Display display;
private Form form;
private Command exit;
private Command process;
private ChoiceGroup gender;
private int currentIndex;
private int genderIndex;
public RadioButtons()
{
display = Display.getDisplay(this);
gender = new ChoiceGroup("Enter Gender", Choice.EXCLUSIVE);
gender.append("Female", null);
currentIndex = gender.append("Male ", null);
gender.setSelectedIndex(currentIndex, true);
exit = new Command("Exit", Command.EXIT, 1);
process = new Command("Process", Command.SCREEN,2);
form = new Form("Gender");
genderIndex = form.append(gender);
form.addCommand(exit);
form.addCommand(process);
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(false);
notifyDestroyed();
}
else if (command == process)
{
currentIndex = gender.getSelectedIndex();
StringItem message = new StringItem("Gender: ",gender.getString(currentIndex));
form.append(message);
form.delete(genderIndex);
form.removeCommand(process);
}
}
}
```


**Figure 6-3.** *Displaying radio buttons on a form*

## DateField Class:

- The DateField class is used to display, edit, or input date and/or time into a MIDlet.

A DateField class is instantiated by specifying a label for the field, a field mode, and a time zone, although time zone is optional.

```
DateField datefield = new DateField("Today", DateField.DATE);
DateField datefield = new DateField("Time", DateField.TIME, timeZone);
```

| Mode | Description |
|---|---|
| DATE | Display, edit, and input a date |
| TIME | Display, edit, and input a time |
| DATE_TIME | Display, edit, and input both date and time |

**Table 6-3.** *DateField Modes*

place a date or time into the date field by calling the setDate() method. The setDate() method requires one parameter, which is an instance of
the Date class containing the date/time value that will appear in the date field.
The getDate() method is called to retrieve the date/time value of the date field.
You can use the date/time value in a number of ways within your MIDlet, such as in a calculation. The setInputMode() method replaces the existing DateField mode with the mode passed as a parameter to the setInputMode() method. The getInputMode() method is used to retrieve the mode of an instance of a DateField.

**Example of Datefield class:**

The instance of the DateField class is placed in the DATE_TIME mode since both date and time are displayed. The date and time is set by passing the Date() construction a date/time value in milliseconds since January 1, 1970. The System.currentTimeMillis() method returns the current time in milliseconds—the number of milliseconds since January 1, 1970.

```
import java.util.*;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class DateToday extends MIDlet implements CommandListener
{
private Display display;
private Form form;
private Date today;
private Command exit;
private DateField datefield;
public DateToday()
{
display = Display.getDisplay(this);
form = new Form("Today's Date");
today = new Date(System.currentTimeMillis());
datefield = new DateField("", DateField.DATE_TIME);
datefield.setDate(today);
exit = new Command("Exit", Command.EXIT, 1);
form.append(datefield);
form.addCommand(exit);
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(false);
notifyDestroyed();
}
}
}
```

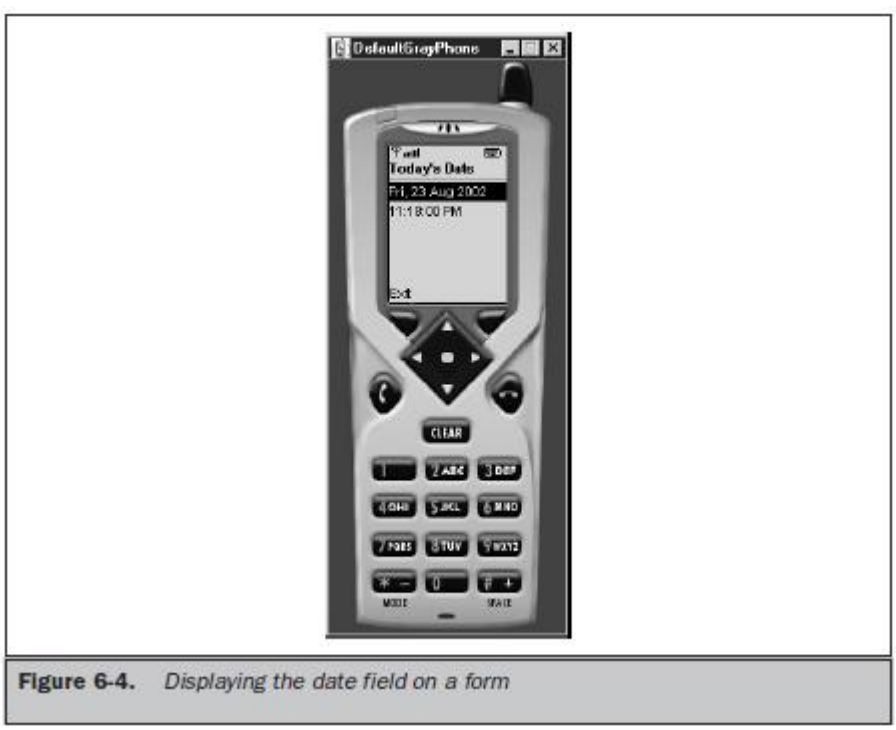



**Figure 6-4.** *Displaying the date field on a form*

Gauge Class:

The Gauge class creates an animated progress bar that graphically represents the status of a process. The indicator on the gauge generated by the Gauge class moves from one end to the other proportionally to the completion of the process measured by the gauge. The Gauge class provides methods to display the gauge and move the indicator. The developer must build the routine into the MIDlet to move the indicator. Each time a calculation is completed, you

must move the indicator one tick. The user of the MIDlet can also control the indicator if the instance of the Gauge class is set in the interactive mode. In interactive mode the user can move the indicator of the gauge to a desired value, such as increasing the volume of a device. The developer must then include a routine in the MIDlet to read the value of the gauge indicator and incorporate the user's input into the MIDlet's processing.
You create an instance of the Gauge class by using the following code segment:
Gauge gauge = new Gauge("Like/Dislike Gauge", true, 100, 0);

This statement creates an interactive gauge with the caption "Like/Dislike Gauge" and a scale of zero to 100. The first parameter passed to the constructor of the Gauge class is a string containing the caption that is displayed with the gauge. The second parameter is a boolean value indicating whether or not the gauge is interactive. The third parameter is the maximum value of the gauge, and the last parameter is the gauge's initial value. can still change the current
value of the gauge indicator by calling the setValue() method. The setValue() method requires one parameter, which is the integer representing the new value. need to determine the current value of the gauge by calling the getValue() method.
The getValue() method returns an integer representing the gauge's current value. can determine the maximum value of the gauge by calling
the getMaxValue() method, which returns the integer representing the current maximum value. If your new value exceeds the maximum value, you can reset the maximum value before setting the new value by calling the setMaxValue() method and passing the method an integer representing the new maximum value. If the Start command was selected, the MIDlet enters the
while loop.
The loop continues as long as the current value of the gauge, as reported by the getValue() method, is less than the maximum value of the gauge returned by the getMaxValue() method. The MIDlet then retrieves the current value again, increments the value by one, and passes the sum as the parameter to the setValue() method, thereby resetting the gauge indicator to show the new status of processing within the while loop. Once the current value is equal to the maximum value of the gauge, the loop is terminated. The Start command is removed from the form, and the setLabel() method is called, resetting the label of the gauge to "Process Completed."

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class GaugeNonInteractive
extends MIDlet implements CommandListener
{
private Display display;
private Form form;
private Command exit;
private Command start;
private Gauge gauge;
private boolean isSafeToExit;
public GaugeNonInteractive()
```

```
{
display = Display.getDisplay(this);
gauge = new Gauge("Progress Tracking", false, 100, 0);
exit = new Command("Exit", Command.EXIT, 1);
start = new Command("Start", Command.SCREEN, 1);
form = new Form("");
form.append(gauge);
form.addCommand(start);
form.addCommand(exit);
form.setCommandListener(this);
isSafeToExit = true;
}
public void startApp()
{
display.setCurrent(form);
}
public void pauseApp()
{
}
```



Listing 6-12
Implementing
a noninteractive
gauge
Listing 6-11
The JAD
file for
Listing 6-12

```
public void destroyApp(boolean unconditional)
throws MIDletStateChangeException
{
if (!unconditional)
{
throw new MIDletStateChangeException();
}
}
public void commandAction(Command command, Displayable displayable)
{
if (command == exit)
{
try
{
destroyApp(isSafeToExit);
notifyDestroyed();
```

```
}
catch (MIDletStateChangeException Error)
{
Alert alert = new Alert("Busy", "Please try again.", null, AlertType.WARNING);
alert.setTimeout(1500);
display.setCurrent(alert, form);
}
}
else if (command == start)
{
form.remove.Command(start);
new Thread(new GaugeUpdater()).start();
}
}
class GaugeUpdater implements Runnable
{
GaugeUpdater()
{
}
public void run()
{
isSafeToExit = false;
try
{
while (gauge.getValue() < gauge.getMaxValue())
{
Thread.sleep(1000);
gauge.setValue(gauge.getValue() + 1);
}
isSafeToExit = true;
gauge.setLabel("Process Completed.");
}
catch (InterruptedException Error)
{
throw new RuntimeException(Error.getMessage());
}
}
}
```


**Figure 6-5.** *Displaying a noninteractive gauge on a form*

Creating an Interactive Gauge:

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class GaugeInteractive extends MIDlet implements CommandListener
{
private Display display;
private Form form;
private Command exit;
private Command vote;
private Gauge gauge;
public GaugeInteractive ()
{
display = Display.getDisplay(this);
gauge = new Gauge("Rate the movie: ", true, 5, 1);
exit = new Command("Exit", Command.EXIT, 1);
vote = new Command ("Vote", Command.SCREEN, 1);
form = new Form("");
form.addCommand(exit);
form.addCommand(vote);
form.append(gauge);
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(false);
notifyDestroyed();
}
else if (command == vote)
{
String msg = String.valueOf(gauge.getValue());
Alert alert = new Alert("Ranking", msg, null, null);
alert.setTimeout(Alert.FOREVER);
alert.setType(AlertType.INFO);
display.setCurrent(alert);
}
}
```


**Figure 6-6.** *Displaying an interactive gauge on a form*

StringItem Class:

AStringItem class is different from other classes derived from the Item class in thata StringItem class does not recognize events. This means that an instance of a StringItem class can never cause an event because the user cannot modify the text of the string item. Other instances of the Item class, such as an instance of the ChoiceGroup class, recognize an event whenever the value of the instance changes, such as selecting a radio button or check box. Although an instance of the StringItem class cannot cause an event to occur, you can modify the instance from within the MIDlet as a result of an event caused by instances of other classes. create an instance of a StringItem class by passing the StringItem class constructor two parameters. The first parameter is a string representing the label of the instance. The other parameter is a string of text that will appear on the screen. You can retrieve the text of the instance of a StringItem class once the instance is created by calling the getText() method. The getText() method returns a string containing the text. Likewise, you can replace the text by calling the setText() method. The setText() method requires one parameter, which is the new text that replaces the current text of the instance. The label of the instance can be changed by calling the setLabel() method. The setLabel() method requires one parameter, which is the replacement label. You can retrieve a label from an instance by invoking the getLabel() method. The getLabel() method returns a string consisting of the label of the instance.

**Example for StringItem:**

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class StringItemExample extends MIDlet
implements CommandListener
{
private Display display;
private Form form;
private StringItem question;
private Command giveup;
private Command exit;
public StringItemExample()
{
display = Display.getDisplay(this);
question = new StringItem("Question: ",
"A plane crashes on the border between Canada
and the US. Where are the survivors buried?");
giveup = new Command("Give Up", Command.SCREEN, 1);
exit = new Command("Exit", Command.EXIT, 1);
form = new Form("Quiz");
form.addCommand(exit);
form.addCommand(giveup);
form.append(question);
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 == giveup)
{
question.setLabel("Answer: ");
question.setText("Survivors are not buried.");
form.removeCommand(giveup);
}
else if (command == exit)
{
destroyApp(false);
notifyDestroyed();
}
}
}
```


**Figure 6-7.** *Displaying a string item on a form*

TextField Class:

The TextField class is used to capture one line or multiple lines of text entered by the user. The number of lines of a text field depends on the maximum size of the text field when you create an instance of the TextField class.

**textfield = new TextField("First Name:", "", 30, TextField.ANY);**

The first parameter is the label that appears when the instance is displayed on the screen. The second parameter is text that you want to appear as the default text for

the instance, which the user can edit. The third parameter is the maximum number of characters that can be held by the instance.

can determine the actual character size of a text box by calling the getMaxSize() method once the text field is instantiated. can also change the maximum size by calling the setMaxSize() method. The setMaxSize() method requires one parameter, which is the new value for the maximum size for the text field. Any time that you need to know the length of the text in the text field you can call the size() method, which returns an integer representing the number of characters existing in the text field. The last parameter passed to the constructor of the TextField class is the constraint (if any) that is used to restrict the type of characters that the user can enter into the text field. Table 6-4 lists the constraints recognized by the TextField class. The instance of the TextField class accepts any character if the ANY constraint is set. You can restrict entry to numeric characters by passing the NUMERIC constraint to the constructor. All non-numeric characters are excluded from the text field. Three special-purpose constraints—EMAILADDR, PHONENUMBER, and URL— act as filters to assure that only valid characters can be entered into the text field for email addresses, phone numbers, and URLs. All other characters are treated as an error and therefore are prevented from being stored in the text field. The PASSWORD constraint can be combined with other constraints to hide characters from being displayed. An asterisk or other character determined by the device is displayed in place of the actual character placed in the text box. The CONSTRAINT_MASK constraint is used to determine the constraint's current value.

There are two methods you can use to retrieve characters entered into a text field by the user of your MIDlet. These are the getString() method and the getChars() method. The getString() method returns the content of the text field as a string, and the getChars() method returns the text field content as a character array. The getChars() method requires that you pass it a character array as a parameter. You place text into a text field by calling either the setString() method or the setChars() method. The setString() method requires one parameter, which is the string containing text that should appear in the text field. The setChars() method requires three parameters. The first is the character array whose data will populate the text field. The second is the position of the first character within the array that will be placed into the text field. The last parameter is the length of characters of the character array that will be placed into the text field. Characters in the character array will replace the entire content of the text field. You can insert characters within the text field without overwriting the entire content of the text field by calling the insert() method. The insert() method has two signatures, one for strings and the other for character arrays. The insert() method used to insert a string into the contents of a text field requires two parameters. The first parameter is the string that will be inserted into the text field. The other parameter is the character position of the current string where the new text is inserted. The text that exists there now will be shifted down to make room for the inserted text. The insert() method used to insert a character array requires four parameters. The first parameter is reference to the array. The second parameter is the position of the first character within the array that will be placed into the text field. The third parameter is the number of characters contained in the array that will be placed into the text field. And the last parameter is the character position of the current text that will be shifted down to make room for the inserted text. Text can be removed from the text field by calling the delete() method, which requires two parameters. The first is the position of the first character to be deleted. The other parameter is the length of characters that are to be deleted. The constraint of a text field can be changed after the instance is

created by calling the setConstraints() method. The setConstraints() method requires you to pass the new constraint as a parameter to the setConstraints() method. You can also determine the current constraint by calling the getConstraints() method. Another sometimes handy method is the getCaretPosition() method. A *caret* is the cursor within the text field, and as you probably guessed, the getCaretPosition() method returns the current position of the cursor. For example, you might design an application that requires the user to select a section of text by positioning the cursor at the first character of the section and then selecting a command. In response, your MIDlet calls the getCaretPosition() method and uses the returned position to extract the section of text from the contents of the text field.

| Constraint | Description |
|---|---|
| CONSTRAINT_MASK | Used to determine the constraint's current value |
| ANY | Input any character |
| EMAILADDR | Input only valid email address characters |
| NUMERIC | Input positive and negative numbers; cannot exclude either positive or negative numbers |
| PASSWORD | Hide input |
| PHONENUMBER | Input characters valid to a phone number sometimes specific to locality and device |
| URL | Input characters valid to a URL |

**Table 6-4.** *TextField Object Constraints*

**Example for Textfield:**

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class HideText extends MIDlet implements CommandListener
{
private Display display;
private Form form;
private Command submit;
private Command exit;
private TextField textfield;
public HideText()
{
display = Display.getDisplay(this);
submit = new Command("Submit", Command.SCREEN, 1);
exit = new Command("Exit", Command.EXIT, 1);
textfield = new TextField("Password:", "", 30, TextField.ANY | TextField.PASSWORD);
form = new Form("Enter Password");
form.addCommand(exit);
form.addCommand(submit);
form.append(textfield);
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 == submit)
{
textfield.setConstraints(TextField.ANY);
textfield.setString("Thank you.");
form.removeCommand(submit);
}
else if (command == exit)
{
destroyApp(false);
notifyDestroyed();
}
}
}
```

# UNIT-IV

**Record Management System:**

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. The nature of the information is application dependent, but typically stretches the breadth of data storage from application settings to information common to a database. Persistence is common to every Java application written in J2SE, J2EE, or J2ME. However, 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. Many small computing devices lack disk drives and access to a network database server or file server, which are the typical resources used for persistence in J2SE and J2EE applications. Therefore, 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. 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 you to store data in columns and rows similar to the organization of data in a table of a database. And you can use RMS to perform the functionality of database management software (DBMS). That is, you can 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 you cannot use SQL to interact with the data. Instead, you'll 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). 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 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. conceptually you can 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:**

They can create multiple record stores as required by your 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.

**Example**:

Let's say that MIDlet A collects information about customers from a sales representative. MIDlet 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.

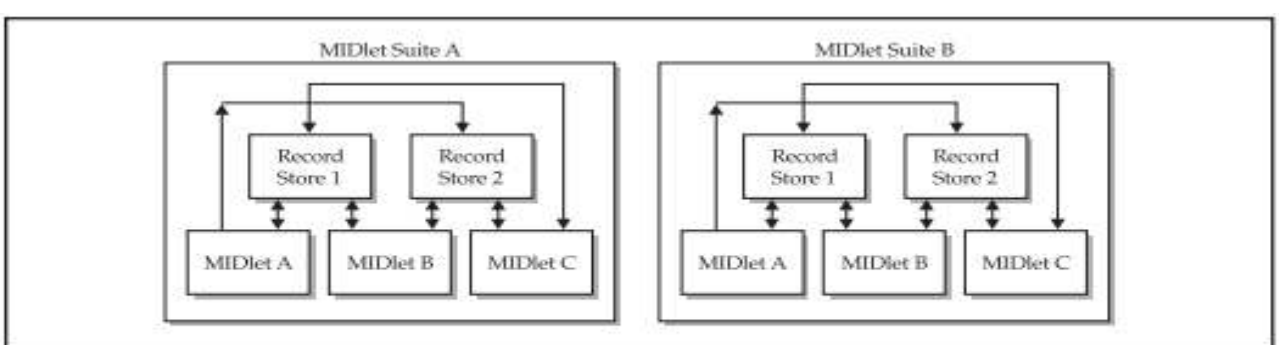



**Figure 8-1.** *MIDlets within MIDlet suite A cannot access record stores collected in MIDlet suite B.*

**Setting Up a Record Store:**

The 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. The 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. Internal resources are utilized to make an open record store available to MIDlets within a MIDlet suite.

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. 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.

**Example for Creating, Opening, Closing, and Removing a Record Store:**

```
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 your 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 you 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.

The 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. 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. 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. 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, as described earlier in this chapter. 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

**recordstore.getRecord(2, myByteArray, 0)**

Example for Creating a New Record and Reading an Existing Record:

import javax.microedition.rms.*; import javax.microedition.midlet.*; import javax.microedition.lcdui.*; import java.io.*; public class WriteReadExample extends MIDlet implements CommandListener { private Display display; private Alert alert; private Form form; private Command exit; private Command start; private RecordStore recordstore = null; public WriteReadExample() { display = Display.getDisplay(this); exit = new Command("Exit", Command.SCREEN, 1); start = new Command("Start", Command.SCREEN, 1); form = new Form("Record"); form.addCommand(exit); form.addCommand(start); form.setCommandListener(this); 306 J2ME: The Complete Reference Complete Reference / J2ME: TCR / Keogh / 222710-9 / Chapter 8 Listing 8-3 Write and read records P:\010Comp\CompRef8\710-9\ch08.vp Thursday, February 06, 2003 11:54:56 AM Color profile: Generic CMYK printer profile Composite Default screen } 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 { String outputData = "First
Record"; byte[] byteOutputData = outputData.getBytes(); recordstore.addRecord(byteOutputData, 0,
byteOutputData.length); } catch ( Exception error) { alert = new Alert("Error Writing", Chapter 8: Record
Management System 307 Complete Reference / J2ME: TCR / Keogh / 222710-9 / Chapter 8 J2ME
DATA MANAGEMENT P:\010Comp\CompRef8\710-9\ch08.vp Thursday, February 06, 2003 11:54:56
AM Color profile: Generic CMYK printer profile Composite Default screen error.toString(), null,
AlertType.WARNING); alert.setTimeout(Alert.FOREVER); display.setCurrent(alert); } try { byte[]
byteInputData = new byte[1]; int length = 0; for (int x = 1; x <= recordstore.getNumRecords(); x++) { if
(recordstore.getRecordSize(x) > byteInputData.length) { byteInputData = new
byte[recordstore.getRecordSize(x)]; } length = recordstore.getRecord(x, byteInputData, 0); } alert = new
Alert("Reading", new String(byteInputData, 0, length), null, AlertType.WARNING);
alert.setTimeout(Alert.FOREVER); display.setCurrent(alert); } catch (Exception error) { alert = new
Alert("Error Reading", 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) 308 J2ME: The Complete
Reference Complete Reference / J2ME: TCR / Keogh / 222710-9 / Chapter 8
P:\010Comp\CompRef8\710-9\ch08.vp Thursday, February 06, 2003 11:54:56 AM Color profile: Generic
CMYK printer profile Composite Default screen { try {
RecordStore.deleteRecordStore("myRecordStore"); } catch (Exception error) { alert = new Alert("Error
Removing", error.toString(), null, AlertType.WARNING); alert.setTimeout(Alert.FOREVER);
display.setCurrent(alert); } } } } }
```


**Figure 8-2.** *Reading a record from a record store and displaying the record on the screen*

# UNIT-V

**Generic connection Framework:**

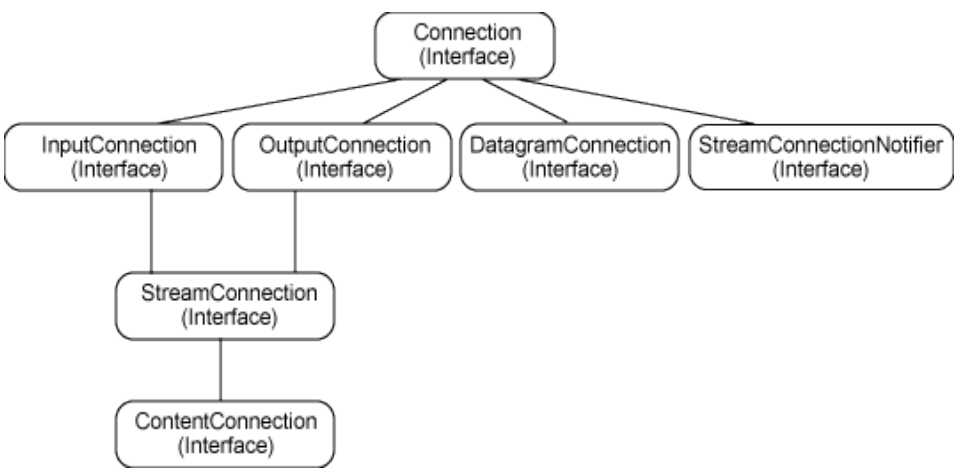
**Network support in J2ME/MIDP**

The Generic Connection Framework (GCF) provides an extensible, generic I/O framework for resource constrained devices. In this final installment in the *J2ME 101* series, author John Muchow walks you through the GCF interfaces, showing you how they facilitate the development and support of various types of network and file I/O on MIDP.

**What is GCF?**

GCF is a set of interfaces defined within the javax.microedition.io package. Figure 1 shows the GCF class hierarchy.

Figure 1. Class hierarchy of the Generic Connect ion Framework




A total of seven interfaces are defined in GCF, with Connection at the root. Notice that both datagram (packet) and stream connections are supported. As you would assume, working your way down the hierarchy you'll find interfaces that provide additional functionality. For example,StreamConnection supports both input and output streams, and ContentConnection extends StreamConnection with support for determining the content type, data length, and encoding format of a stream.

The Connector class is used to open every type of connection in GCF. Here you can see the format for the open() method inside the Connector class:

**Connector.Open("protocol:address;parameters");**

**Connection protocol support:**

GCF is exceptionally flexible in its support for different connection protocols. When a request is made to open a connection, the Connectorclass uses its Class.forName() method to search for a class that implements the requested protocol. If found, an object is returned that implements the Connection interface, show above figure

In Listing 1, you can see the code to open various connection types.

**Opening a connection**

**Listing 1**. Opening various connection types

```
Connector.Open("socket://www.corej2me.com.com:55");
Connector.Open("http://www.corej2me.com");
Connector.Open("datagram://www.corej2me.com:1000");
Connector.Open("file://makefile.txt");
```

GCF has a total of seven methods for creating a connection. All of them are shown in Listing 2.

**Listing 2**. Seven methods to create a connection

```
  Connector  (public class Connector)
  public static Connection open(String name)
  public static Connection open(String name)
  public static Connection open(String name, int mode, boolean timeouts)
  public static DataInputStream openDataInputStream(String name)
  public static DataOutputStream openDataOutputStream(String name)
  public static InputStream openInputStream(String name)
  public static OutputStream openOutputStream(String name)
```

Listing 3 illustrates one way to open a connection and read data through a stream.

**Listing 3** . One way to open a connection

```
// Create a ContentConnection
String url = "http://www.corej2me.com"
ContentConnection connection = (ContentConnection) Connector.open(url);

// With the connection, open a stream
InputStream iStrm = connection.openInputStream();

// ContentConnection includes a length method
int length = (int) connection.getLength();
if (length != -1)
{
  byte imageData[] = new byte[length];

  // Read the data into an array
  iStrm.read(imageData);
}
```

The ContentConnection class isn't the only option we have for creating a connection. We could also choose to create an InputStreamdirectly. In Listing 4, you can see what might happen if we needed to download an image over a network connection and create an Imagebased on the downloaded contents.

**Listing 4**. Creating an input stream directly

```
InputStream iStrm = (InputStream) Connector.openInputStream(url);
Image img = null;

try
{
  ByteArrayOutputStream bStrm = new ByteArrayOutputStream();

  int ch;
  while ((ch = iStrm.read()) != -1)
    bStrm.write(ch);

  // Place into image array
  byte imageData[] = bStrm.toByteArray();

  // Create the image from the byte array
  img = Image.createImage(imageData, 0, imageData.length);
}
finally
{
  // Clean up
  if (iStrm != null)
    iStrm.close();
}
```

As you may have noticed, bypassing the ContentConnection leaves us no method to determine the length of the incoming data. This isn't much of a problem, however, because we can use the ByteArrayOutputStream to read and transfer the data to our destination array.

**The DownloadImage MIDlet:**

Let's write a short MIDlet that builds on the code shown in Listing 4. The DownloadImage MIDlet will demonstrate the steps to download and display an image in an MIDP application. The MIDlet will use a ByteArrayOutputStream to download the remote data and then display the resulting image on a Form using an ImageItem.

Take a look at the complete code for the DownloadImage_MIDlet and then we'll discuss it in more detail.

Once the MIDlet is running the main user interface should appear in your WTK device emulator, as shown in the figures below. Figure 2 shows a TextBox that prompts for the URL of image. Clicking the *View* command initiates the download.

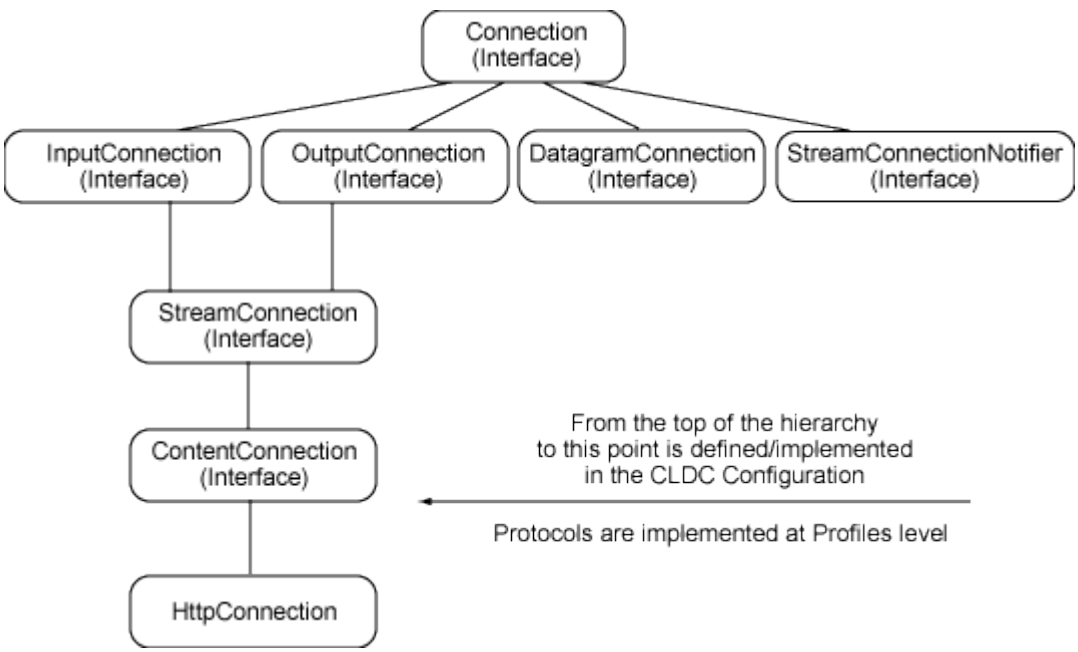
Figure 2. A textbox displaying a URL prompt


Once the image is received it is displayed on the device as shown in Figure 3.

Figure 3. An example image display screen

**HTTP support in MIDP**

Now that you've seen how GCF supports various types of connections and we've developed our first networking MIDlet, it's time to take a closer look at HTTP support in MIDP. We'll start with an updated hierarchy diagram that indicates which class provides support for HTTP connections.




The original MIDP 1.0 specification only required that devices support the HTTP connection protocol, whereas the more recent MIDP 2.0 spec requires support for both HTTP and HTTPS, with the latter offering support for more secure network connections. The APIs to work with these protocols are HttpConnection and HttpConnections, respectively. In addition to these mandated protocols, a device manufacturer can choose to support additional communication protocols such as datagrams or sockets. While at times convenient, you should know that using vendor-specific protocols could affect your application's portability to other devices.

**Cookies and Session Tracking**

HTTP is a stateless protocol where each request andπ response pair is a separate conversation Sometimes, you want the server to remember who youπ are: this can be done using a session When the client sends an HTTP request to the server itπ should include a session ID - the server can use this session ID to identify the client The server can use this session ID to do useful stuff likeπ retrieve preferences or maintain a shopping cart The most common way to store a session ID on the clientπ side is using HTTP cookies A cookie is a little piece of data that is passed from theπ server to client and then back to the server at the next request .