

# **J.B.Institute of Engineering and Technology**

(UGC Autonomous)

(Accredited by NAAC, Approved by AICTE & Permanently Affiliated to JNTUH)

**Department of Information Technology**



## **DATA STRUCTURES THROUGH C**

Lecture Notes  
II year I Semester  
(2018-2019)

Prepared By  
**B.Deepthi Reddy**  
Assistant Professor

# MOBILE APPLICATION DEVELOPMENT

## 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.
  - ❖ 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:

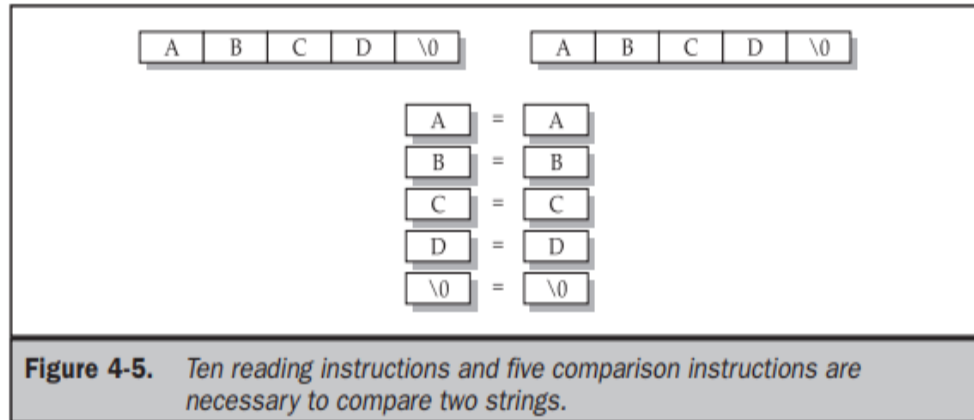
- . Best practices are proven design and programming techniques used to build J2ME systems. Patterns are routines that solve common programming problems that occur in such systems.
- **Keep Applications Simple:**
  - u must adapt to a new mind-set when creating applications for small computing devices because of limited resources available and the inability to easily expand resources to meet application requirements
- **Keep Applications Small:**
  - 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:**
  - 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:**
  - 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:**
  - 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 receiver. Cellular telephone networks use technology that attempts to maintain connection as the mobile device moves from one cell to another cell.
- **Simplify the User Interface:**
  - 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:**
  - 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:**
  - **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

## Mobile Application Development Notes

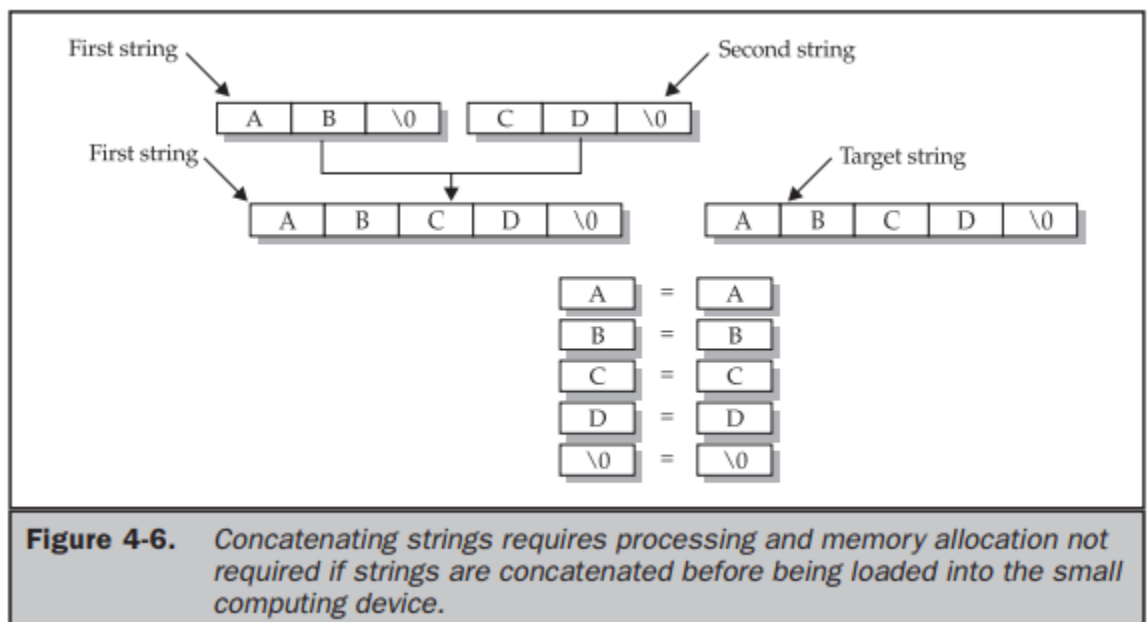
character of each string into the CPU for comparison. This process continues until either the null character is reached or a letter pair is different. T

- he entire process might require ten reading instructions and five comparison instructions, depending on when a mismatch is discovered.



- **Avoid Synchronization:**

- 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.



- 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:**
  - 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:**
  - 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:**
  - 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:**
  - 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:**
  - 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:**
  - 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:**
  - 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:**

- 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:**

- 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 the user 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's Display 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)
{
```

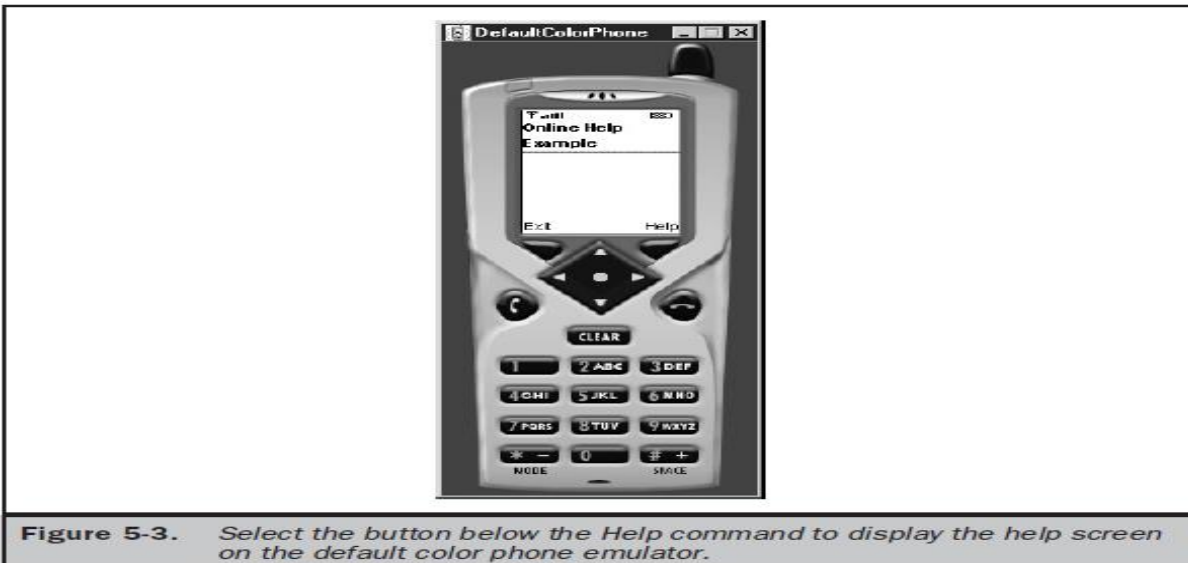
```
    if (command == cancel)
    {
        destroyApp(false);
        notifyDestroyed();
    }
}
```

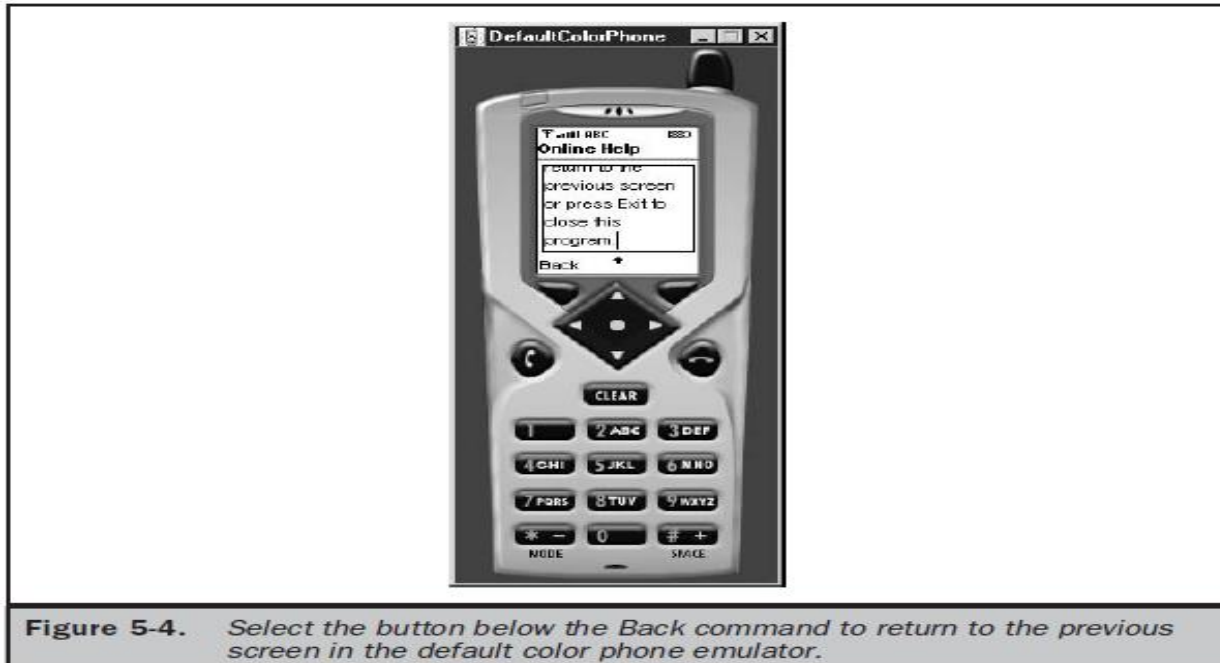
### **Example:**

```
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 5-4.** 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:

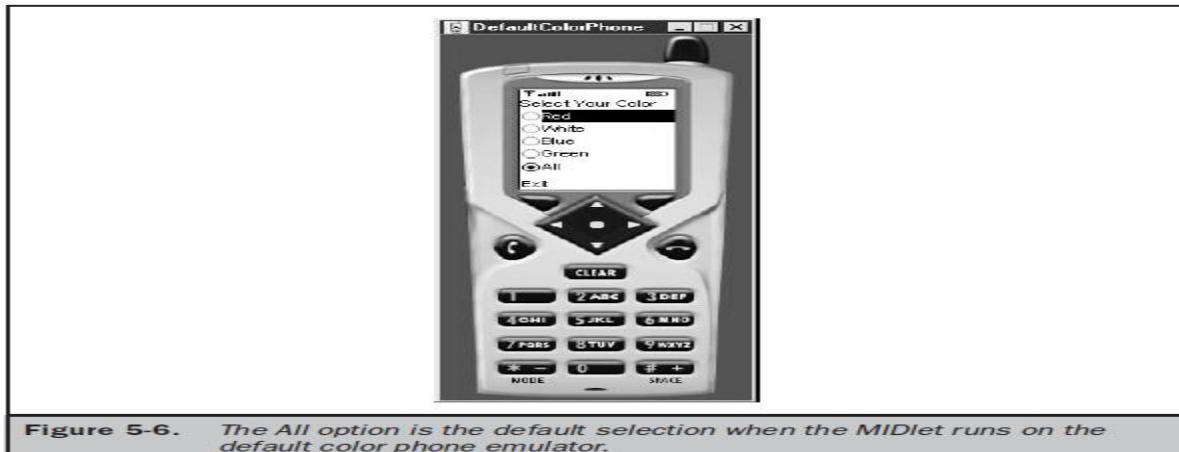
- Each MIDlet that utilizes instances of the Item class within a form must have an **itemStateChanged()** method to handle state changes in these instances.
- The **itemStateChanged()** method, contains one parameter, which is an instance of the Item class.
- The instance passed to the **itemStateChanged()**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



### 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 used to temporarily reject a request from the application manager either to start the MIDlet (**startApp()**) or to destroy the MIDlet (**destroyApp()**).
- A MIDletStateChangeException cannot be thrown within the **pauseApp()** method.

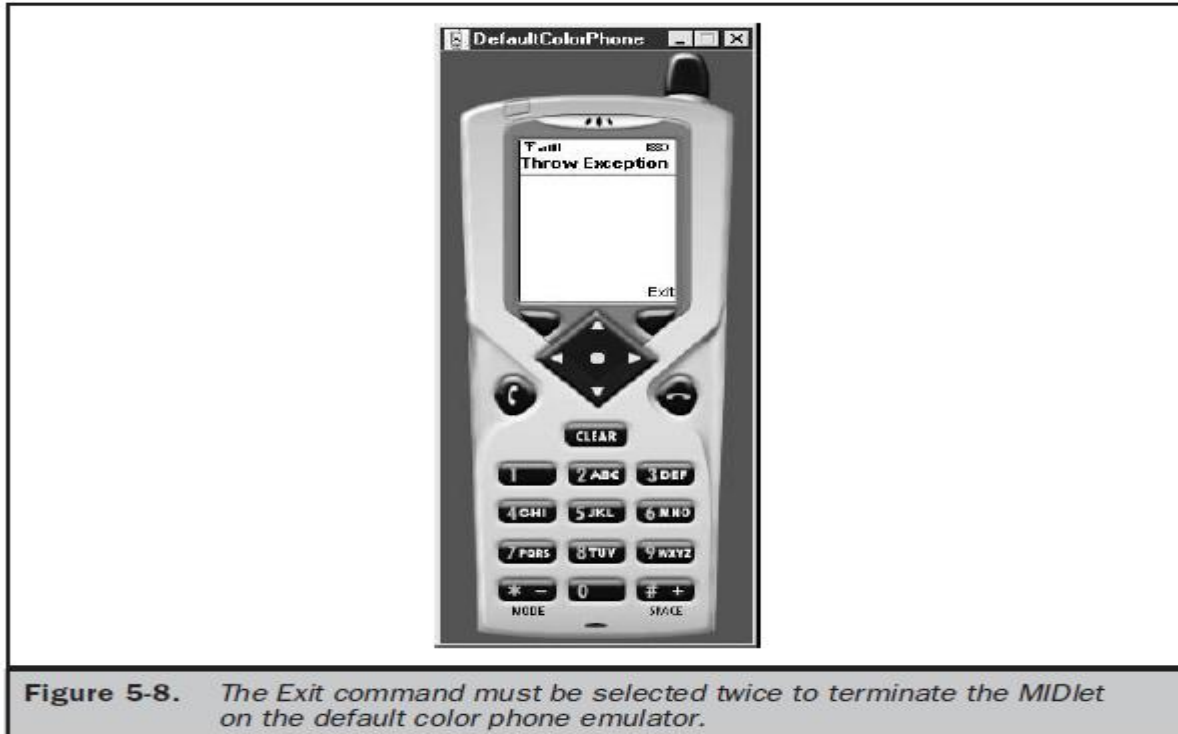
### Throwing a MIDletStateChangeException:

```
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.





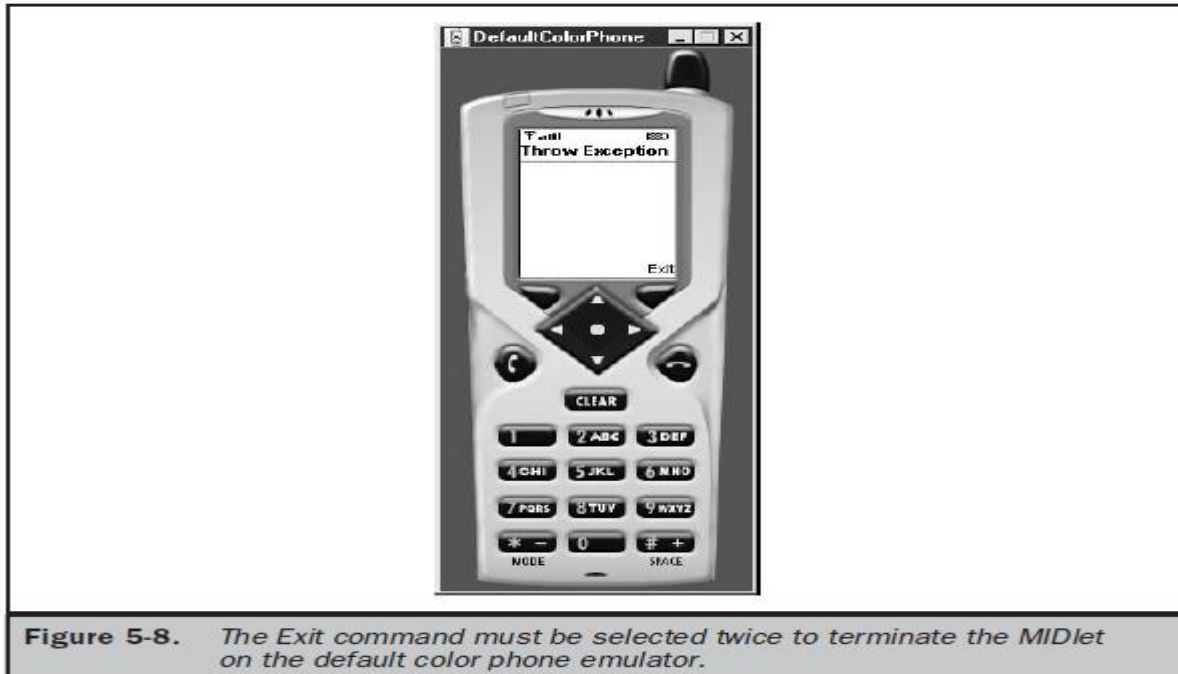
- isSafeToQuit variable that is used to indicate whether it is safe to terminate the MIDlet. In the constructor, the isSafeToQuit is assigned a false, implying that the MIDlet should not be terminated.
- Likewise in the constructor there are statements that obtain instances to the Display class, Command 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;
}
}
}
```



### The Palm OS Emulator:

- The ROM 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 to download 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.

## MOBILE APPLICATION DEVELOPMENT

### 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 a modal alert dialog box, or pass a time value in milliseconds indicating time to terminate the alert 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 the integer 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 once an 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.**

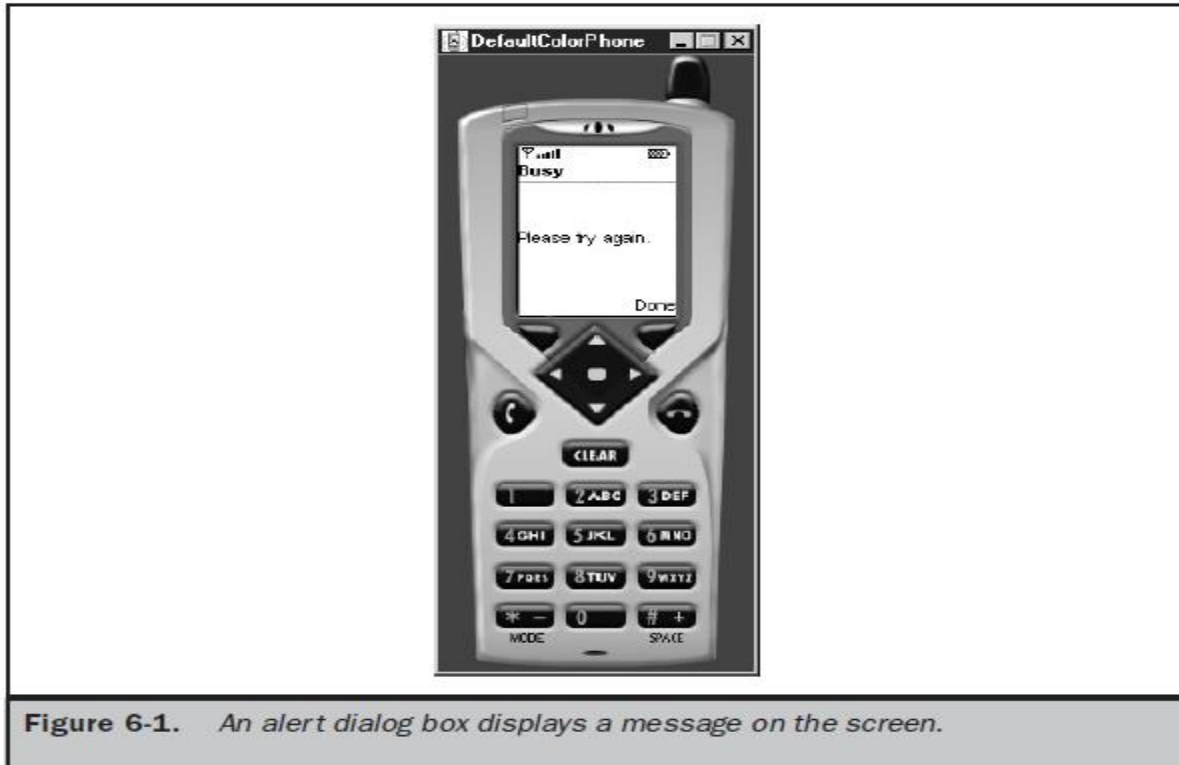
```
form = new Form("Communication Link");
alert = new Alert("Failure", "Lost communication link!", null, null);
alert.setTimeout(Alert.FOREVER);
display.setCurrent(alert);
display.setCurrent(form);
```

### 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 instance of 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 typically implement 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, beginning with 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 created and 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 two parameters.** 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
    }
}
```

### ChoiceGroup 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 **ChoiceGroup 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.
- The **getSelectedFlags()** 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();
}
}
```



}

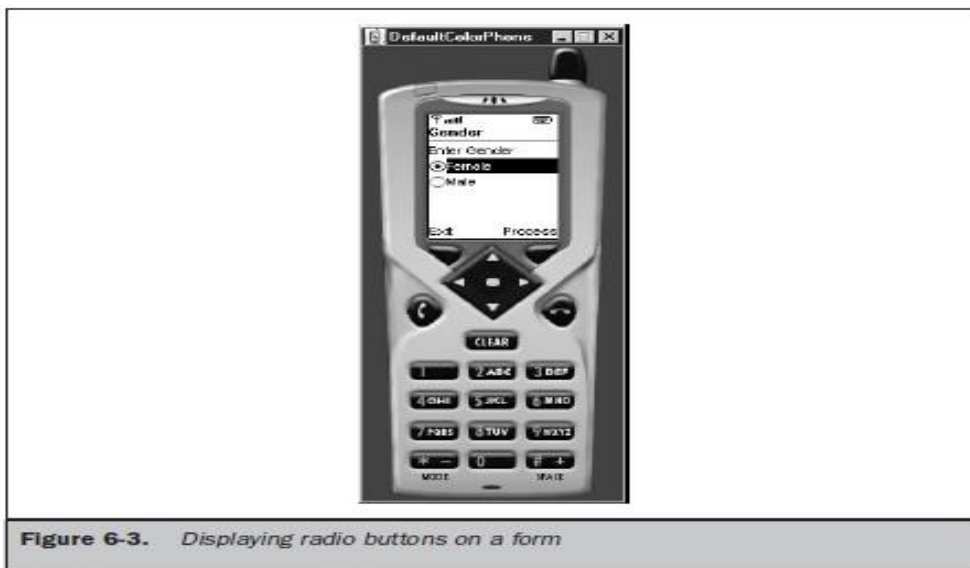
the type is **EXCLUSIVE**, which limits selection to only one set of choices within the group. set a default selection by first storing the index number of the Male radio button that is returned by the **append()** method.

- 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.

### 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);
    }
}
}
```



### 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:

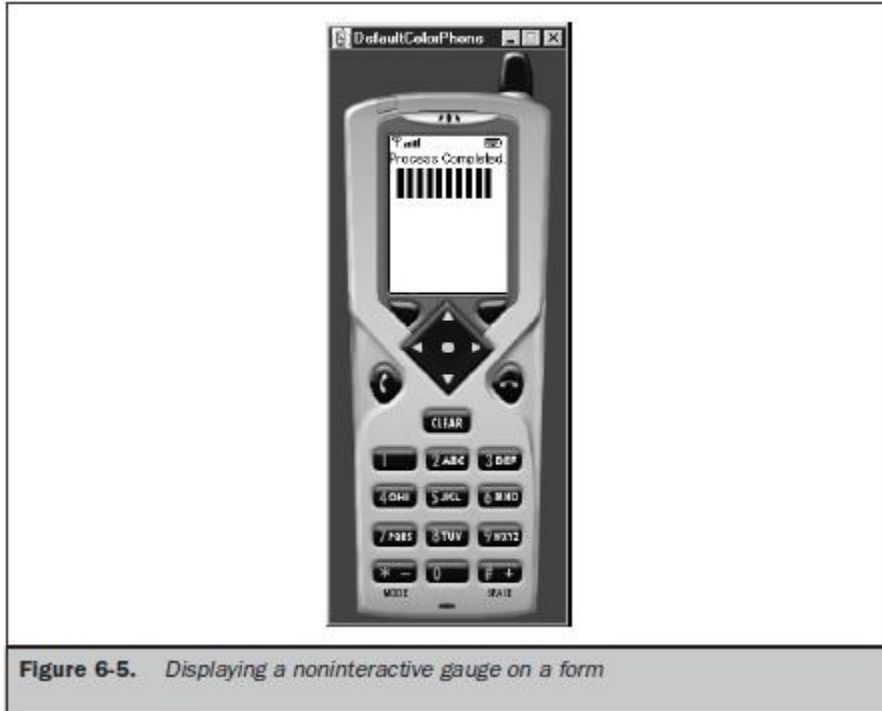
- The **Gauge class** creates an animated progress bar that graphically represents the status of a process.
- The indicator on the gauge generated by the Gauge class moves from one end to the other proportionally to the completion of the process measured by the gauge.
- The Gauge class provides methods 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()
{
}
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());
        }
    }
}
```



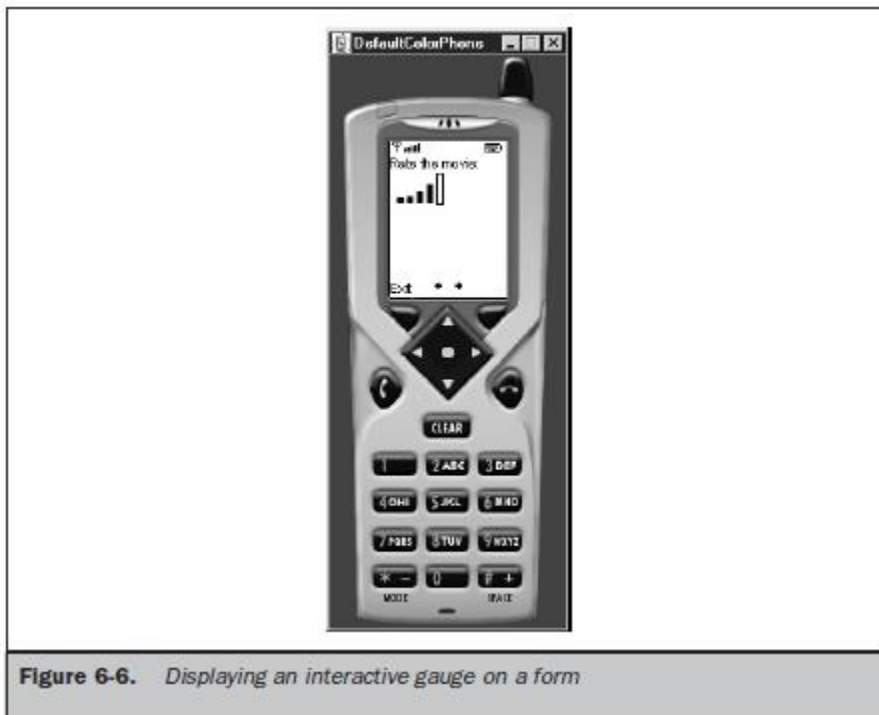
**Figure 6-5.** *Displaying a noninteractive gauge on a form*

### 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*

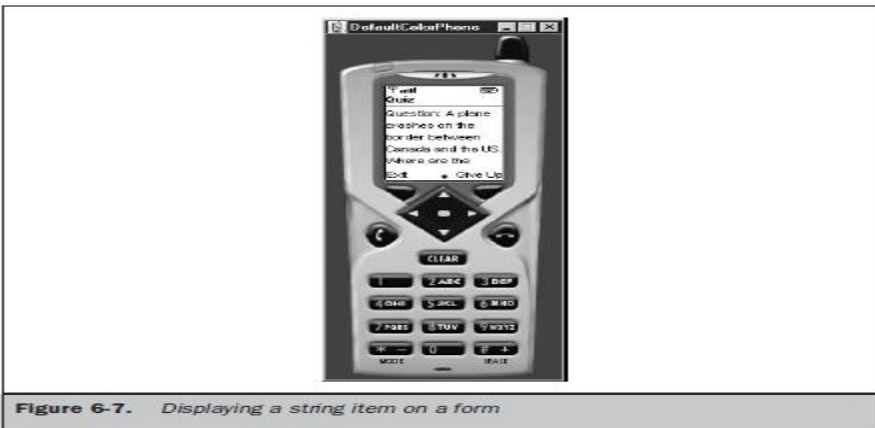
### StringItem Class:

- AStringItem class is different from other classes derived from the Item class in that a 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.
- we 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();
}
}
}
```



### 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 textfield 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.
- The instance of the TextField class accepts any character if the ANY constraint is set.
- 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.
- 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.
- 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.

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();  
    }  
}
```

### ImageItem Class:

- **There are two types of images that can be displayed.**
- These are **immutable images and mutable images**.
- **An *immutable image*** is loaded from a file or other resource and cannot be modified once the image is displayed. Icons associated with MIDlets are immutable images.
- **A *mutable image*** is drawn on the screen using methods available in the Graphics class.
- Once drawn, your MIDlet can redraw any portion of the image.
- An immutable image is drawn on a screen, and a mutable object is drawn on a canvas.
- Mutable images are displayed using the Graphics class, which is derived from the Canvas class.
- The first step in displaying an immutable image is to create an instance of the Image class by calling the **createImage() method**.
- The **createImage() method** requires **one parameter** that contains the name of the file containing the image.
- **Make sure that you include the full path to the file in the parameter.**
- **The next step is to create an instance of the ImageItem class. The constructor of the ImageItem class requires four parameters.**
- The **first** is a string that becomes the label for the image.
- The **second** parameter is reference to the instance of the Image class created in step one.
- The **third parameter** is the layout directive.
- The **last parameter** is a string referred to as alternate text that is displayed in place of the image if for some reason the image cannot be displayed by the device. Some applications won't require you to specify a label or alternate text; therefore, use a null as the value of the parameter in place of a string.

Value	Description
LAYOUT_DEFAULT	Use the device's default layout

LAYOUT_LEFT	Place image left
LAYOUT_RIGHT	Place image right
LAYOUT_CENTER	Center image
LAYOUT_NEWLINE_BEFORE	Start a new line and then draw the image
LAYOUT_NEWLINE_AFTER	Draw the image and then start a new line

- Can modify the image layout after the image is displayed on the screen by calling the **getLayout() method** and **setLayout() method**.
- The **getLayout()** method returns the current layout directive of an instance of an ImageItem.
- The **setLayout() method** replaces the current layout with a new layout whose directive is passed as a parameter to the setLayout() method.
- can modify the label and alternate text associated with an image by calling the **getLabel()** method, **setLabel()** method, **getAltText()** method, and **setAltText()** method.
- The **getLabel()** and **getAltText()** methods retrieve the current label and alternate text, and the **setLabel()** and **setAltText()** methods are called to replace them.
- Both the **setLabel()** and **setAltText()** methods require **one parameter**, which is the replacement text.
- The image itself is replaceable by calling the **getImage()** method and the **setImage()** method.
- The **getImage()** method fetches the current image associated with the instance of the ImageItem, and the **setImage()** method associates a new image with the instance.
- The **setImage()** method requires you to pass it an instance of an Image class.

### Creating an Instance of an ImageItem Class:

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class ImmutableImage extends MIDlet
implements CommandListener
{
    private Display display;
    private Form form;
    private Command exit;
    private Image image;
    private ImageItem imageItem;
    public ImmutableImage()
    {
        display = Display.getDisplay(this);
        exit = new Command("Exit", Command.EXIT, 1);
        form = new Form("Immutable Image Example");
        form.addCommand(exit);
        form.setCommandListener(this);
        try
        {
            image = Image.createImage("myimage.png");
```



```
imageItem = new ImageItem(null, image,
ImageItem.LAYOUT_NEWLINE_BEFORE |
ImageItem.LAYOUT_LEFT |
ImageItem.LAYOUT_NEWLINE_AFTER, "My Image");
form.append(imageItem);
}
catch (java.io.IOException error)
{
Alert alert = new Alert("Error", "Cannot load myimage.png.",
null, null);
alert.setTimeout(Alert.FOREVER);
alert.setType(AlertType.ERROR);
display.setCurrent(alert);
}
}
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();
}
}
}
```

### List Class:

- The List class is used to display a list of items on the screen from which the user can select one or multiple items.
- There are **three formats for the List class**:
  - radio buttons,
  - check boxes, and an
  - implicit list that does not use a radio button or check box icon

- A List class differs from the ChoiceGroup class by the way events of each instance are handled by a MIDlet. As you recall from the discussion about the ChoiceGroup class, an ItemStateListener is used to listen to events generated by an instance of a ChoiceGroup class.
- Those events are then passed along to the itemStateChanged() method for processing. Likewise, a commandAction() method is used to process command events
- In contrast, a list does not generate an item state change event; therefore, a Command needs to be added to initiate processing.
- a command event is automatically generated when the user selects an item from an instance of an implicit formatted List class.
- Typically, an implicit formatted List class is used to create a menu.
- The commandAction() method is automatically called to process the menu selection without requiring the user to select a command to process the selection.
- A List class is derived from the Screen class and does not require a container. In contrast, the ChoiceGroup class is derived from the Item class and requires an instance of a Form class to contain the instance of the ChoiceGroup class.
- You can create an instance of the List class with or without list items.
- An instance is created without list items by passing the constructor of the **List class two parameters**.
- The **first parameter** is a string that contains the titles of the list, and
- the **other parameter** is the format of the list commonly referred to as the listType.
- can include list items when creating the instance of a List class by **passing two additional parameters to the List class constructor**.
- The **first two parameters** are title and listType.
- The **third parameter** is a string array whose elements contain list items that can be selected by the user of your MIDlet.
- The **fourth parameter** is an array of instances of the Image class, each associated with a corresponding list item.
- List items can be added to an instance of a List object by calling the **append() method or insert() method**.
- The **append() method** requires **two parameters**.
- The **first parameter** is the string that contains the new list item, and
- the **second parameter** is an instance of the Image class of an image that is associated with the new list item.
- The new list item is appended to the end of the list.
- The **insert() method** is very similar in design to the append() method, except the new list item is inserted within the list.
- **Three parameters are necessary for the insert() method**.
- The **first parameter** is the index number of the list item above which the new list item is inserted.
- The other **two parameters** are the same as the parameters of the **append() method**.
- can retrieve the list item selected by the user by calling the **getSelectedIndex()** method.
- The getSelectedIndex() method returns the index number of the selected list item.

- pass the returned index number as the parameter to the **getString() method**, which returns the string of the selected list item, which is then processed by the **commandAction() method**.
- If the instance of the List class is a check box, then call the **getSelectedFlag() method**.
- The **getSelectedFlag() method requires one parameter**, which is a **boolean array**.
- The method then populates the boolean array with the selected flag value of each list item.
- The **size() method** returns the number of items on the list and can be used to set the size of the boolean array.
- One or more list items can become the default selection by calling the **setSelectedIndex() method or the setSelectedFlags() method**.
- The **setSelectedFlags() method is used to set the selected flag of one list item and requires two parameters**.
- The **first parameter** is the index number of the list item being selected, and the other parameter is a boolean value, where true signifies that the list item is selected and false signifies unselected.
- The **setSelectedIndex() method** performs an operation similar to the **setSelectedFlags()**, except the **setSelectedIndex()** sets the selected status for all list items.
- The **setSelectedIndex() requires one parameter**, which is a boolean array containing the selected status for the entire list.
- Any list item can be replaced by calling the **set() method**.
- The **set() method requires three parameters**.
- The **first** is the index number of the list item being replaced.
- The **second** parameter is the string replacing the string of the specified list item.
- And the **last parameter** is an Image object that contains the image associated with the replacement list item.
- A list item can be removed from the list by calling the **delete() method**.
- The **delete() method requires one parameter, which is the index number of the list item being deleted**.

### Creating an Instance of a List Class:

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class ListImplicit extends MIDlet implements CommandListener
{
    private Display display;
    private List list;
    private Command exit;
    Alert alert;
    public ListImplicit()
    {
        display = Display.getDisplay(this);
        exit = new Command("Exit", Command.EXIT, 1);
```

```
list = new List("Menu:", List.IMPLICIT);
list.append("New",null);
list.append("Open",null);
list.addCommand(exit);
list.setCommandListener(this);
}
public void startApp()
{
display.setCurrent(list);
}
public void pauseApp()
{
}
public void destroyApp(boolean unconditional)
{
}
public void commandAction(Command command, Displayable displayable)
{
if (command == List.SELECT_COMMAND)
{
String selection = list.getString(list.getSelectedIndex());
alert = new Alert("Option Selected", selection, null, null);
alert.setTimeout(Alert.FOREVER);
alert.setType(AlertType.INFO);
display.setCurrent(alert);
}
else if (command == exit)
{
destroyApp(false);
notifyDestroyed();
}
}
}
```



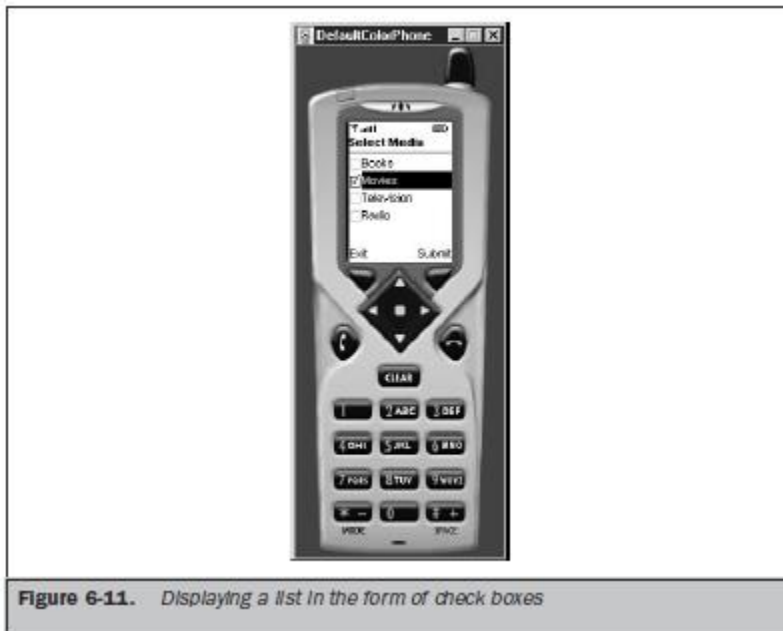
**Figure 6-10.** Displaying an implicit list on the screen

### Creating an Instance of a Check Box–Formatted List Class:

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class ListCheckBox extends MIDlet implements CommandListener
{
    private Display display;
    private Command exit;
    private Command submit;
    private List list;
    public ListCheckBox()
    {

        display = Display.getDisplay(this);
        list = new List("Select Media", List.MULTIPLE);
        list.append("Books", null);
        list.append("Movies", null);
        list.append("Television", null);
        list.append("Radio", null);
        exit = new Command("Exit", Command.EXIT, 1);
        submit = new Command("Submit", Command.SCREEN, 2);
        list.addCommand(exit);
        list.addCommand(submit);
        list.setCommandListener(this);
    }
    public void startApp()
    {
        display.setCurrent(list);
    }
    public void pauseApp()
    {
    }
    public void destroyApp(boolean unconditional)
    {
    }
    public void commandAction(Command command, Displayable Displayable)
    {
        if (command == submit)
        {
            boolean choice[] = new boolean[list.size()];
            StringBuffer message = new StringBuffer();
            list.getSelectedFlags(choice);
            for (int x = 0; x < choice.length; x++)
            {
                if (choice[x])
                {
                    message.append(list.getString(x));
                    message.append(" ");
                }
            }
        }
    }
}
```

```
}  
}  
Alert alert = new Alert("Choice", message.toString(),  
null, null);  
alert.setTimeout(Alert.FOREVER);  
alert.setType(AlertType.INFO);  
display.setCurrent(alert);  
  
list.removeCommand(submit);  
}  
else if (command == exit)  
{  
destroyApp(false);  
notifyDestroyed();  
}  
}  
}
```



**Figure 6-11.** *Displaying a list in the form of check boxes*

### Creating an Instance of a Radio Button–Formatted List Class:

```
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;  
public class ListRadioButtons extends MIDlet implements CommandListener  
{  
private Display display;  
private Command exit;  
private Command submit;  
private List list;  
public ListRadioButtons()
```

```
{
display = Display.getDisplay(this);
list = new List("Select one", List.EXCLUSIVE);
list.append("Male", null);
list.append("Female", null);
exit = new Command("Exit", Command.EXIT, 1);
submit = new Command("Submit", Command.SCREEN, 2);
list.addCommand(exit);
list.addCommand(submit);
list.setCommandListener(this);
}
public void startApp()
{
display.setCurrent(list);
}
public void pauseApp()
{
}
public void destroyApp(boolean unconditional)
{
}
public void commandAction(Command command, Displayable Displayable)
{
if (command == submit)
{
Alert alert = new Alert("Choice",
list.getString(list.getSelectedIndex()),
null, null);
alert.setTimeout(Alert.FOREVER);
alert.setType(AlertType.INFO);
display.setCurrent(alert);
list.removeCommand(submit);
}
else if (command == exit)
{
destroyApp(false);
notifyDestroyed();
}
}
}
```

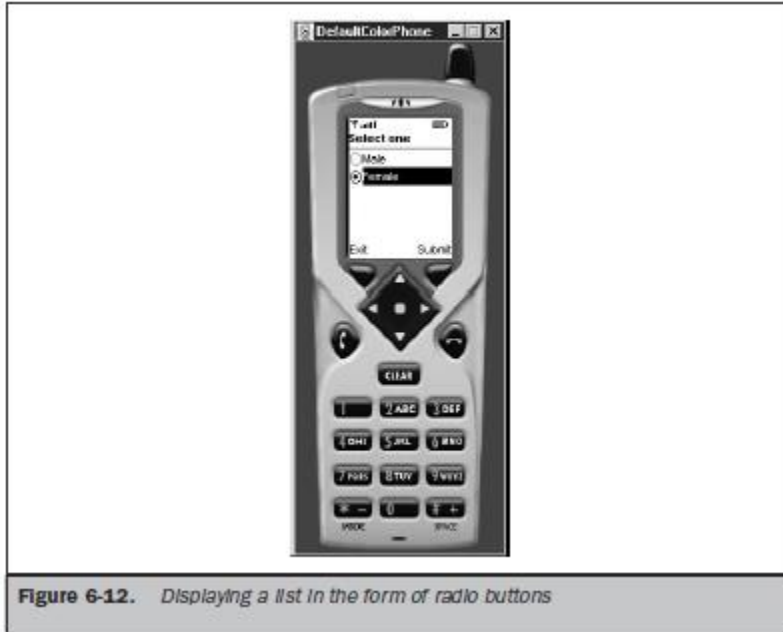


Figure 6-12. Displaying a list in the form of radio buttons

### TextBox Class:

- The TextBox class is very similar to a TextField class. Both are used to receive multiple lines of textual data from a user and constrain text that can be entered using the constraint directives .
- can request that a maximum number of characters be allowed in instances of both the TextBox class and the TextField class.
- Characters that exceed the display area of the screen become scrollable in many devices.
- **The TextBox class and TextField class differ in that the TextBox class is derived from the Screen class, while the TextField class is derived from the Item class.**
- This means that an instance of the Form class cannot contain an instance of the TextBox class, while an instance of a TextField class must be contained within an instance of the Form class.
- Another important **difference between the TextBox class and the TextField class** is that the TextBox class uses a CommandListener and cannot use an ItemStateListener.
- An ItemStateListener is used with an instance of the TextField class, although many times the content of an instance of the TextField class is retrieved and processed when the user selects a command associated with a form that contains the text field.
- An instance of the **TextBox class is created by passing four parameters** to the TextBox class constructor.
- The **first parameter** is the title of the text box.
- The **second parameter** is text used to populate the instance.
- The **third parameter** is the maximum number of characters that can be entered into the instance. Keep in mind that this parameter is a request and may not be fulfilled by the



device. The device determines the maximum number of characters for an instance of the `TextBox` class.

- The **last parameter** is the constraint used to limit the types of characters that can be placed within the instance.
- The `TextBox` class has the same methods as found in the `TextField` class.

### Creating an Instance of a `TextBox` Class

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class TextBoxCapture extends MIDlet implements CommandListener
{
    private Display display;
    private TextBox textbox;
    private Command submit;
    private Command exit;
    public TextBoxCapture()
    {
        display = Display.getDisplay(this);
        submit = new Command("Submit", Command.SCREEN, 1);
        exit = new Command("Exit", Command.EXIT, 1);
        textbox = new TextBox("First Name:", "", 30, TextField.ANY);
        textbox.addCommand(exit);
        textbox.addCommand(submit);
        textbox.setCommandListener(this);
    }
    public void startApp()
    {
        display.setCurrent(textbox);
    }
    public void pauseApp()
    {
    }
    public void destroyApp(boolean unconditional)
    {
    }
    public void commandAction(Command command, Displayable displayable)
    {
        if (command == submit)
        {
            textbox.setString("Hello, " + textbox.getString());
            textbox.removeCommand(submit);
        }
        else if (command == exit)
        {
            destroyApp(false);
        }
    }
}
```

```
notifyDestroyed();  
}  
}  
}
```



Figure 6-13. Displaying a text box on the screen

### Ticker Class:

- The Ticker class is used to scroll text horizontally on the screen much like a stock ticker scrolls stock prices across the screen.
- An instance of the Ticker class can be associated with any class derived from the Screen class and be shared among screens.
- An instance of a Ticker object is created by passing the constructor of the Ticker class a string containing the text that is to be scrolled across the screen.
- Cannot control the location on the screen where scrolling occurs. Likewise, there is no control over the speed of the scrolling.
- The device that runs the MIDlet controls both location and speed.
- can retrieve the text associated with an instance of the Ticker class by calling the **getString() method**.
- Can replace the text currently scrolling across the screen by calling the **setString() method**.
- The **setString() method requires one parameter, which is a string containing the replacement text**.

### Creating an Instance of a Ticker Class:

```
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;  
public class TickerList extends MIDlet implements CommandListener
```

```
{
private Display display;
private List list;
private final String tech;
private final String entertain;
private Ticker ticker;
private Command exit;
private Command submit;
public TickerList()
{
display = Display.getDisplay(this);
tech = new String ("IBM 55 MSFT 32 SUN 52 CISCO 87");
entertain = new String ("CBS 75 ABC 455 NBC 243 GE 21");
exit = new Command("Exit", Command.SCREEN, 1);
submit = new Command("Submit", Command.SCREEN, 1);
ticker = new Ticker(tech);
list = new List("Stock Ticker", Choice.EXCLUSIVE);
list.append("Technology", null);

list.append("Entertainment", null);
list.addCommand(exit);
list.addCommand(submit);
list.setCommandListener(this);
list.setTicker(ticker);
}
public void startApp()
{
display.setCurrent(list);
}
public void pauseApp()
{
}
public void destroyApp(boolean unconditional)
{
}
public void commandAction(Command command, Displayable display)
{
if (command == exit)
{
destroyApp(true);
notifyDestroyed();
}
else if (command == submit)
{
if (list.getSelectedIndex() == 0)
{
ticker.setString (tech);
```

```
}  
else  
{  
ticker.setString(entertain);  
}  
}  
}  
}
```



## LOW-LEVEL DISPLAY

### Canvas:

- Today's small computing devices are capable of running form-based applications, games that challenge the best of us, and applications that interact with remote computers.
- Practically any application you can imagine can be designed to operate within the confines of a small computing device.
- Many applications that you develop will use a high-level user interface, usually thought of as an object that handles its own display and consists of lists, radio buttons, check boxes, images, and text.
- Classes associated with the high-level interface handle the pixel level detail necessary to draw radio buttons, check boxes, and other objects on the screen.
- basically call methods and let the methods handle display. Occasionally, you may be called upon to create an application that sizzles and wows the user with fancy graphics and animation.
- To get the sizzle and wow into the application, need to go beyond the high-level user interface and get down and dirty into the pixel level of your application, where you control the position of every picture element that appears on the screen. **This is referred to as the *low-level user interface*.**

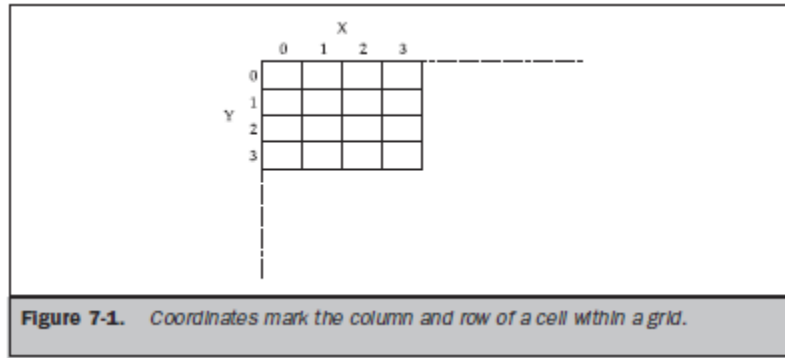
### The Canvas

- Each MIDlet has one instance of the Display class, and the Display class has one derived class called the Displayable class.
- Everything a MIDlet displays on the screen is created by an instance of a class that is derived from the displayable class.
- The Display class hierarchy is

```
public class Display
public abstract class Displayable
public abstract class Screen extends Displayable
public abstract class Canvas extends Displayable
public class Graphics
```
- **The Displayable class has two subclasses:**
  - Screen and
  - Canvas.
- The **Screen class** and its derivatives are used to create high-level components.
- The **Canvas class** and its derivatives are used to gain low-level access to the display, which is necessary for graphic- and animation-based applications.
- A graphic is used with a canvas.
- An instance of the Canvas class as an artist's canvas on which you draw images that might include text.
- An instance of the Graphics class is similar to the artist's tools that are used to draw an image. For example, color, lines, and arcs are some of the graphic tools available to create an image on the canvas.
- The **Canvas class and the Graphics class** give you pixel control over everything that appears on the canvas.
- This low-level control is particularly noticeable whenever text is placed on the canvas because you control every aspect of how characters are formed to display the text.
- displaying text using the Graphics class requires you to specify the height,width, and other characteristics that describe how each character of the text is to be drawn on the screen.

### The Layout of a Canvas

- The canvas is divided into a virtual grid in which each cell represents one pixel.
- Coordinates mark the column and row of a cell within the grid .
- The x coordinate represents the column, and the y coordinate represents the cell's row.
- The first cell located in the upper-left corner of the grid has the coordinate location of 0, 0, where the first zero is the x coordinate and the other zero is the y coordinate.
- the size of the canvas is device dependent since canvas size and the screen size are the same.
- The screen size of a mobile telephone might be different from the screen size of a PDA, and yet both devices are capable of running the same MIDlet.
- MIDlet should determine the canvas size of the device that implements your graphic application before drawing on the screen. The canvas size is measured in pixels.
- MIDlet should determine the canvas size of the device by calling the getWidth() and getHeight() methods of the Canvas class.



### Proportional Coordinates:

- The values (in pixels) returned by the **getWidth()** and **getHeight()** methods can be used to draw an image at a given location that is proportional to the size of the canvas by using relative coordinates rather than exact coordinates on the canvas.
- Let's say that the **first element** of the image you want drawn on the canvas is located in the center of the canvas.
- **calculate the center coordinate based on the return value of the getWidth() and getHeight() methods.**

### Example:

Assume the size of the canvas is 200 pixels wide (columns) and 200 pixels high (rows). The coordinate of the center of the canvas is 99, 99 (remember that coordinates are zero based), calculated as:

$x = \text{getWidth()} / 2$

$y = \text{getHeight()} / 2$

- If you knew the size of the canvas, you could plot each pixel that is required to draw an image.
- The image will be symmetrical within the screen. However, the symmetry is disrupted when the size of the canvas changes and the image size remains the same.
- If the screen size is 200 pixels wide by 200 pixels high, the starting coordinate of the line is 1, 1 (the cell in the second column and second row beginning at the left corner).
- The ending coordinate is 1, 198.
- The starting coordinates are set as specific values because the line always begins one pixel from the left and top of the canvas. Likewise, the x coordinate will always be the second row and can be fixed at row coordinate one.
- However, the column coordinate used to specify the termination of the line must be calculated so that the line always appears one pixel away from the right end of the canvas regardless of the size of the canvas.

$\text{startX} = 1$

$\text{startY} = 1$

$\text{endX} = 1$

$\text{endY} = \text{getWidth()} - 1$

### The Pen

- An image is drawn on a canvas using a virtual pen. Using a virtual pen is very similar to using a real pen to draw an image on paper.
- That is, the pen is dropped on the canvas at a specified coordinate, filling the cell with the color of ink used in the pen.
- Cells change from their present color to the color of the ink as the pen is repositioned on the canvas.
- For example, a horizontal line forms on the canvas when the virtual pen is dragged horizontally across the canvas.
- Likewise, dragging the virtual pen vertically down the canvas draws a vertical line.
- A **virtual pen** is used by instances of the Graphics class to draw rectangles, arcs, and other graphical image components on the canvas. You don't directly create and use a virtual pen.

### Painting:

- Graphical components used to create an image on a canvas are drawn on the canvas when the **paint() method** of the Displayable class is called. This is referred to as **painting**.
- The **paint() method** is an abstract method that is used both by instances and derivatives of the Screen class and Canvas class.
- The contents of the paint() method are statements that draw images on the screen.
- Derivatives from the Screen class have two predefined methods used to paint the screen.
- Images painted by derivatives of the Screen class (Textbox, List, Alert, and Form) are radio buttons, check boxes, list, text, and other constructs that you find in a graphical user interface.
- The **first predefined method is paint()**, which contains instructions that set parameters for drawing an image, such as defining the virtual pen.
- The **other method is paintContent()**, which is called at the end of the paint() method and contains statements to actually draw the image.
- The developer doesn't become directly involved with the paint() method or the paintContent() method when building an application that uses the high-level user interface because details on how to display images are already defined by the derivative of the Screen class.
- The **paint() method requires one parameter, which is reference to the instance of the Graphics class created by your application.**

```
protected void paint(Graphics graphics)
{
    graphics.drawRect(12, 6, 40, 20));
}
```

### The paint() and repaint() Methods

- Cannot call the paint() method directly. Instead, the paint() method is called automatically by the setCurrent() method when the MIDlet is started.
- call the repaint() whenever the canvas or a portion of the canvas must be refreshed.
- There are **two versions of the repaint() method**.

- **One version requires no parameters** and repaints the entire canvas.
- The **other version requires four parameters** that define the region of the canvas that is to be repainted.
- The **first two parameters** are the x and y coordinates for the upper-left corner of the region, and
- the **last two parameters** are the width and height of the region.
- The **repaint() method** is capable of repainting only the portion of the frame that changed rather than the entire frame, which dramatically reduces the time that is necessary to change a frame on the screen.
- The **serviceRepaints() method** is another painting method that you'll use when developing a low-level user interface for your application.
- A paint request is one of many requests a MIDlet can make to the application manager of a small computing device.
- Other requests can be made to store data or to communicate with a remote computer.
- The **serviceRepaints() method** directs the device's application manager to override outstanding requests for service with the repaint request.
- The repaint request becomes the next request to be processed by the application manager.

### showNotify() and hideNotify()

- The device's application manager calls the **showNotify()** method immediately before the application manager displays the canvas.
- the **showNotify()** method with statements that prepare the canvas for display, such as initializing resources by beginning threads or assigning values to variables as required.
- The **hideNotify()** method is called by the application manager after the canvas is removed from the screen.
- The **hideNotify()** method with statements that free resources that were allocated when the **showNotify()** method was called.
- This includes deactivating threads and resetting values assigned to variables as necessary.

### User Interactions

- One of **two techniques** can be used to receive user input into your low-level J2ME application.
- The **first technique** is to create one or more instances of the Command class.
- Once an instance of a command is created, the instance is associated with the instance of the Canvas class by calling the addCommand() method.
- If you associate a command with a canvas, need to associate a CommandListener to the canvas in order to monitor command events generated by the user selecting a command.
- The other technique is to use low-level user input components that generate low-level user events.
- **These components are key codes, game actions, and pointers.**
  - A **key code** is a numerical value sent by the small computing device when the user of your application selects a particular key. Each key on the device's keypad



is identified by a unique key code.

■ A **game action** is a keystroke that a person uses to play a game on the small computing device. MIDP defines a set of constants that represent keystrokes common to game controllers.

■ A **pointer event** is input received from a pointer device attached to the small computing device, such as a touch screen or mouse.

### Working with Key Codes

- Each key on an ITU-T keypad, which is used on cellular telephones, is mapped to a standard set of key codes.
- J2ME associates key code values with constants; however, use the constant instead of the constant value.
- All small computing devices that use the ITU-T keypad adhere to these key codes.
- Some of these devices also have other keys on the keypad, each of which is also assigned a key code.
- The manufacturer's specification for the device usually lists key codes used by the device for keys outside of those on the standard ITU-T keypad.
- There are three empty methods that are called when a particular key event occurs while your MIDlet is running.
- These methods are **keyPressed()**, **keyReleased()**, and **keyRepeated()**.
- The **keyPressed()** method is called by the application manager whenever a key is pressed by the user.
- the **keyReleased()** method is called when the key selected by the user is released.
- And the **keyRepeated()** method is called by the application manager when the user holds down the key, causing the key to be automatically repeated.
- MIDlet can inquire whether or not the repeated key feature is supported by calling the **hasRepeatEvents()**
- **All three methods require one parameter**, which is an integer that represents the value of the key code passed to the method by the device's application manager.
- An if statement or switch case statement is used to compare the incoming key code with key code constants that are processed by the MIDlet.

Constant	Value
KEY_NUM0	48
KEY_NUM1	49
KEY_NUM2	50
KEY_NUM3	51
KEY_NUM4	52
KEY_NUM5	53
KEY_NUM6	54
KEY_NUM7	55
KEY_NUM8	56
KEY_NUM9	57
KEY_STAR	42
KEY_POUND	35

**Table 7-1.** Key Code Constants and Key Code Values

### Detecting and Processing Key Codes

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class KeyCodeExample extends MIDlet
{
    private Display display;
    private MyCanvas canvas;
    public KeyCodeExample ()
    {
        display = Display.getDisplay(this);
        canvas = new MyCanvas(this);
    }
    protected void startApp()
    {
        display.setCurrent(canvas);
    }
    protected void pauseApp()
    {
    }
    protected void destroyApp( boolean unconditional )
    {
    }
    public void exitMIDlet()
    {
        destroyApp(true);
        notifyDestroyed();
    }
}
class MyCanvas extends Canvas implements CommandListener
```

```
{
private Command exit;
private String direction;
private KeyCodeExample keyCodeExample;
public MyCanvas (KeyCodeExample keyCodeExample)
{
direction = "2=up 8=dn 4=lt 6=rt";
this.keyCodeExample = keyCodeExample;
exit = new Command("Exit", Command.EXIT, 1);
addCommand(exit);
setCommandListener(this);
}
protected void paint(Graphics graphics)
{
graphics.setColor(255,255,255);
graphics.fillRect(0, 0, getWidth(), getHeight());
graphics.setColor(255, 0, 0);
graphics.drawString(direction, 0, 0,
Graphics.TOP | Graphics.LEFT);
}
public void commandAction(Command command, Displayable displayable)
{
if (command == exit)
{
keyCodeExample.exitMIDlet();
}
}
protected void keyPressed(int key)
{
switch ( key ){
case KEY_NUM2:
direction = "up";
break;
case KEY_NUM8:
direction = "down";
break;
case KEY_NUM4:
direction = "left";
break;
case KEY_NUM6:
direction = "right";
break;
}
repaint();
}
}
```



**Figure 7-2.** A menu is displayed prompting the user to select a menu option.



**Figure 7-3.** The name of the direction selected by the user is drawn on the canvas.

### Working with Game Actions:

- The theme may differ among computer games, but the way players interact with a game is fairly constant across all computer games.
- Players can move up, down, left, right, and they can fire. Typically, directional movement causes a game piece to move in a corresponding direction or changes the viewpoint of the player, depending on the nature of the game.
- Directional movement and fire are referred to as game actions, and MIDP game action defines constants that enable you to utilize game actions within your MIDlet without being concerned about the appropriate key code that is assigned to each action.
- Each game action is associated with one or more keys on the keypad.
- For example, the down game action might be associated with a down directional key and a number on the keypad. Pressing either key causes the same game action to occur.

- Each key can be assigned to only one game action. This means pressing the down game action key doesn't also generate an up game action.
- **A game action causes the keyPressed() method, keyReleased() method, and keyRepeated() method to be called, depending on the key pressed by the player.**
- Can detect which game action occurred by calling the **getGameAction()** method.
- **The getGameAction() method requires one parameter—the key code of the key selected by the player**
- **which is passed as a parameter to the keyPressed(), keyReleased(), or keyRepeated() method.**
- An if statement or a switch case statement can be used to compare the incoming key code to game action constants.
- **Each game action constant is a data member of the Canvas class and is referenced by using the name of the game action constant, such as Canvas.LEFT, Canvas.RIGHT, Canvas.UP, Canvas.DOWN, and Canvas.FIRE.**
- **There are two alternative ways to detect the game action key selected by the player. The first is to compare key code values by calling the getKeyCode() method.**
- **The getKeyCode() method requires one parameter, which is the name of the game action constant.**
- **The getKeyCode() returns the key code value associated with the game action constant that can then be directly compared to the incoming key code value passed to the keyPressed(), keyReleased(), or keyRepeated() method.**

```
        if (getKeyCode(FIRE) == keycode)
        {
            //fire
        }
```
- The other way to determine the player's selection is to retrieve the name of the key that is associated with the incoming key code by calling the **getKeyName()** method.
- **The getKeyName() method requires one parameter, which is the key code value. The getKeyName() method returns the name of the key represented by the key code value.**

```
if (getKeyName(getKeyCode(FIRE).equals(getKeyName(keycode))))
{
    //fire
}
```

Game Action Constant	Description	Game Action Constant Value
UP	Move up	1
DOWN	Move down	6
LEFT	Move left	2
RIGHT	Move right	5
FIRE	Fire	8
GAME_A	Device defined	9
GAME_B	Device defined	10
GAME_C	Device defined	11
GAME_D	Device defined	12

**Table 7-2.** *Game Action Constants*

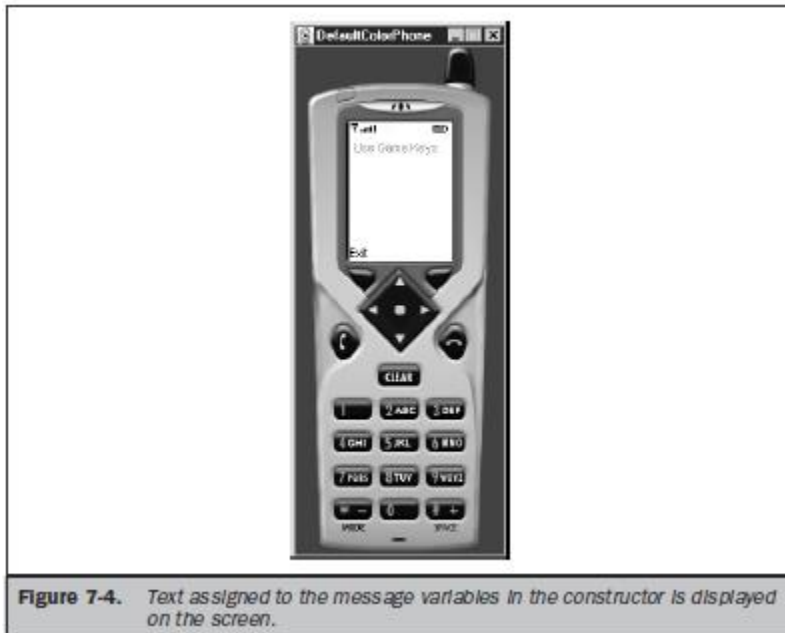
### Detecting and Processing Game Actions:

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class GameActionExample extends MIDlet
{
    private Display display;
    private MyCanvas canvas;
    public GameActionExample()
    {
        display = Display.getDisplay(this);
        canvas = new MyCanvas (this);
    }
    protected void startApp()
    {
        display.setCurrent(canvas);
    }
    protected void pauseApp()
    {
    }
    protected void destroyApp( boolean unconditional )
    {
    }
    public void exitMIDlet()
    {
        destroyApp(true);
        notifyDestroyed();
    }
}
class MyCanvas extends Canvas implements CommandListener
{
    private Command exit;
```

```
private String message;
private GameActionExample gameActionExample;
private int x, y;
public MyCanvas (GameActionExample gameActionExample)
{
    x = 5;
    y = 5;
    direction = "Use Game Keys";
    this.gameActionExample = gameActionExample;
    exit = new Command("Exit", Command.EXIT, 1);
    addCommand(exit);
    setCommandListener(this);
}
protected void paint(Graphics graphics)
{
    graphics.setColor(255,255,255);
    graphics.fillRect(0, 0, getWidth(), getHeight());
    graphics.setColor(255, 0, 0);
    graphics.drawString(message, x, y, Graphics.TOP | Graphics.LEFT);
}
public void commandAction(Command command, Displayable displayable)
{
    if (command == exit)

    {
        gameActionExample.exitMIDlet();
    }
}
protected void keyPressed(int key)
{
    switch ( getGameAction(key) ){
    case Canvas.UP:
        message = "up";
        y--;
        break;
    case Canvas.DOWN:
        message = "down";
        y++;
        break;
    case Canvas.LEFT:
        message = "left";
        x--;
        break;
    case Canvas.RIGHT:
        message = "right";
        x++;
    }
```

```
break;  
case Canvas.FIRE:  
    message = "FIRE";  
    break;  
}  
repaint();  
}  
}
```



### Working with Pointer Devices:

- A pointer device is something other than a keyboard or keypad that is used to interact with an application.
- The most commonly used pointer devices are a touch screen and a mouse, although you can be sure that new pointer devices are bound to find their way into the marketplace in the future.
- A pointer event occurs whenever the person uses a pointer device to interact with your MIDlet.
- There are **three pointer events that your MIDlet must process**. These are when the person **presses a pointer device, releases a pointer device, and drags a pointer device**.
- A person presses a pointer device by applying pressure to a portion of a touch screen or by clicking the mouse button. This causes a **press event**.
- A **release event** occurs once pressure is removed from the touch screen or the mouse button.
- And your MIDlet is notified of a **drag event** whenever the person moves the pointer device during a press event.
- three methods that are automatically called by the device's application manager when a pointer event occurs.



- These methods are the **pointerPressed() method, the pointerReleased() method, and the pointerDragged() method.**
- All **three methods require two parameters.**
- The **first parameter** is an integer representing the x coordinate of the pointer device, and the other parameter is an integer representing the y coordinate. Typically, your MIDlet will use these parameters to change the image that appears on the screen.

### Detecting and Processing Pointer Events:

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class PointerExample extends MIDlet
{
    private Display display;
    private MyClass canvas;
    public PointerExample()
    {
        display = Display.getDisplay(this);
        canvas = new MyClass (this);
    }
    protected void startApp()
    {
        display.setCurrent( canvas );
    }
    protected void pauseApp()
    {
    }
    protected void destroyApp( boolean unconditional )
    {
    }
    public void exitMIDlet()
    {
        destroyApp(true);
        notifyDestroyed();
    }
}
class MyClass extends Canvas implements CommandListener
{
    private Command exit;
    private Command erase;
    private boolean eraseFlag = false;
    private boolean isFirstPaint;
    private int sX = 0,sY = 0, cX = 0, cY = 0;
    private PointerExample pointerExample;
    public MyClass (PointerExample pointerExample)
```

```
{
this.pointerExample = pointerExample;
exit = new Command("Exit", Command.EXIT, 1);
erase = new Command("Erase", Command.SCREEN, 1);
addCommand(exit);
addCommand(erase);
setCommandListener(this);
isFirstPaint = true;
}
protected void paint(Graphics graphics)
{
if (eraseFlag || isFirstPaint)
{
graphics.setColor(255, 255, 255);
graphics.fillRect(0, 0, getWidth(), getHeight());
eraseFlag = isFirstPaint = false;
sX = 0;
sY = 0;
cX = 0;
cY = 0;
return;
}
graphics.setColor(0, 0, 0);
graphics.drawLine(sX, sY, cX, cY);
sX = cX;
sY = cY;
}
public void commandAction(Command command, Displayable displayable)
{
if (command == exit)
pointerExample.exitMIDlet();
else if (command == erase)
{
eraseFlag = true;
repaint();
}
}
protected void pointerPressed(int x, int y)
{
sX = x;
sY = y;
}
protected void pointerDragged(int x, int y)
{
cX = x;
cY = y;
}
```



```
repaint();  
}  
}
```

### Graphics:

- The screen of the low-level user interface is a canvas, which is an instance of the Canvas class. The canvas is organized into a grid in which each cell of the grid is a pixel.
- An image is drawn on the canvas by using a virtual graphical device called a **graphic context**, such as the rectangle and line.
- A **graphic context** is an instance of the Graphics class.
- Reference to the graphic context is passed to the paint() method.
- A mutable image, is an image that can be altered by your MIDlet.
- Reference to the graphic context passed to a paint() method exists for the duration of the paint() method. Once the MIDlet leaves the paint() method, the graphic context goes out of scope.
- The graphic context can no longer be used to draw on the canvas, even if reference to the graphic context is retained.
- In contrast, a graphic context created in association with a mutable image remains available to the MIDlet as long as reference to the image and the image itself remains in scope.

### Stroke Style and Color

#### Stroke Style:

- Every **graphic context** has two characteristics can control from within the MIDlet. **These are stroke style and color.**
- **Stroke style** defines the appearance of lines used to draw an image on the canvas, and color specifies the **background and foreground color of the image.**
- can use **two kinds of stroke styles** when drawing images on the canvas:
  -  **solid and**
  -  **dotted.**
- As the names imply, **the solid stroke style causes the graphic context to use a solid line when drawing the image.**
- The **dotted stroke style results in the image being drawn using a dotted line.** The solid stroke style is the default.
- Skipping pixels along the lines of the image creates the dotted stroke.
- The small computing device determines the number of pixels skipped.
- You cannot modify the appearance of the dotted stroke, and you might discover that skipped pixels may affect the appearance of the image.
- For example, a pixel at the corner of a rectangle might be missing and therefore ruin the illusion of a square-cornered rectangle.

- Calling the **setStrokeStyle() method** determines the stroke style that will be used by a graphic context. A stroke style setting is particular to each graphic context and does not affect other graphic contexts.
- The **setStrokeStyle() method requires one parameter**, which is a constant that represents a stroke style. There are **two constants, SOLID and DOTTED**, both of which are members of the Graphics class.
- This method returns an integer that can be compared within your MIDlet to the stroke style constants.

### Colors:

- Combining degrees of red, green, and blue creates the foreground and background color of a graphic context.
- The degree of each color is specified as an integer value within the range of 0 to 255. Zero produces the darkest possible value of the color, and 255 produces the lightest possible value.
- For example, color values 0, 0, 0 (red, green, blue) produce black, and color values 255, 255, 255 produce white.
- Values between these extremes produce shades of various colors. All integers in Java are 32 bits. Of those 32 bits, 8 bits are used to represent red, blue, and green.
- All color values are stored in one integer. The 8 highest order bits are not used.
- The **isColor() method** returns a **boolean value** that is true if color is supported; otherwise a false value is returned, indicating that the device supports the gray scale instead of color.
- The **numColors() method** returns an integer representing the number of colors or shades of gray supported by the device. can use both of these return values to reset a color choice for the graphic context that is appropriate for the colors available on the small computing device.
- Set the color of a graphic context by calling the **setColor() method** of the Graphics class.
- The **setColor() method requires either one parameter or three parameters** depending on how you represent your choice of color.
- A color can be represented as one integer or three integers, where each of the three integers represents a color value of **red, green, and blue**.
- Let's take a closer look at how color values are represented in order to understand the technique used to represent a color by a single integer. Remember that the highest order bits are not used to represent color values. The color is represented by the next 24 bits.
- The **24 bits are divided into three 8-bit groups**.
- The **first 8-bit group** represents the color value of **red**.
- The **second 8-bit group** represents the color value of **green**.
- And the **third 8-bit group** represents the color value of **blue**.
- The bitwise shift operator (<<) and the bitwise OR operator (|) are then used to insert each of the color values into the appropriate place within the 24 bits that represent the color.
- Let's use this example to illustrate. Suppose we use the following color values:

- red = 50, green = 200, and blue = 150. You can set the color of a graphic context by calling `setColor(50, 200, 150)`. You can also pass the combined values of red, green, and blue to the `setColor()` method. Here's how these values are combined and passed to the `setColor()` method:

**`setColor((50 << 16) | (200 << 8) | 150);`**

Here are the decimal values for each color:

<b>Red</b>	<b>50</b>	00110010
<b>Green</b>	<b>200</b>	11001000
<b>Blue</b>	<b>150</b>	10010110

- When these values are combined into one 32-bit value they appear as:  
**00000000001100101100100010010110**
- There are **two techniques** available for determining the current color setting of a graphic context.
- They involve determining the value of each component of the color (red, green, blue) either by calling the appropriate method for each component or by masking the color value.
- The simplest way to determine the color is to call the **`getRedComponent()` method**, **`getGreenComponent()` method**, and **`getBlueComponent()` method**.
- Each returns an integer representing the color value of the corresponding component.
- An **alternative technique** is to retrieve the 32-bit color value by calling the **`getColor()` method**, then using a bit mask to extract each component of the color.
- The `getColor()` method returns the color value setting of the instance of the Graphics class, which is the graphic context.
- The **first statement** masks all but the first 8 bits of the color value, which represents the red component of the color.
- The **second statement** masks all but the second 8 bits of the color value. This is the green component.
- And the **last statement** masks the last 8 bits of the color, which is the blue component.  





```
red = graphics.getColor() & 0x00ff0000;  
red = red >> 16;  
green = graphics.getColor() & 0x0000ff00;  
green = green >> 8;  
blue = graphics.getColor() & 0x000000ff;
```

### Lines:

- Lines are drawn on the canvas by calling the **`drawLine()` method**.
- The **`drawLine()` method** creates a line from a starting coordinate to an ending coordinate.
- **Four parameters are required by the `drawLine()` method**.
- The **first two parameters** are integers representing the starting x, y coordinate of the line.
- The other **two parameters** are integers representing the ending x, y coordinate of the line.
- The thickness of the line, referred to as the *weight*, is typically measured in point size, where zero is the thinnest possible line.

- Unfortunately, you cannot easily change the weight of a line drawn on the canvas because the weight is always one pixel.
- The only way to create a heavier (thicker) line is to draw multiple, abutting lines, which appear as one thicker line on the screen.

### Rectangles:

- A rectangle is an area of the canvas defined by four corners, just like a rectangle that you can draw using paper and pencil.
- Define a rectangle's dimensions by identifying coordinates for the upper-left corner and the lower-right corner.
- **Four types of rectangles** can be drawn on a canvas.
- These are
  -  an outlined rectangle,
  -  filled rectangle,
  -  outlined rectangle with rounded corners, and
  -  a filled rectangle with rounded corners.
- An outlined rectangle is one where only line segments connecting the corners are drawn.
- The inside of the rectangle remains the same color as the outside of the rectangle.
- The filled rectangle also draws line segments to connect corners, but the inside of the rectangle is filled with the same color as the drawn line segments.
- Create an **outlined rectangle by calling the drawRect() method** and
- The **filled rectangle by calling the fillRect() method**.
- **Both methods create a square-cornered rectangle and require four parameters.**
- The **first two parameters** are the coordinates of the upper-left corner of the rectangle (x1, y1), and
- The **last two parameters** are the width and height of the rectangle (x2, y2).
- The color used to draw a rectangle must be set using the setColor() method before drawing the rectangle.
- Otherwise the current color of the graphic context is used both to color the outline and fill the inside of the rectangle, depending on the type of rectangle being drawn.
- Creating a round-cornered rectangle is nearly identical to creating a square-cornered rectangle, except you must specify the horizontal and vertical diameter of the arc used to create the round corners.
- The **horizontal diameter is referred to as the arc width**, and the **vertical diameter is referred to as the arc height**.
- The diameter represents the sharpness of the corner, where the smaller the diameter, the sharper the corner appears.
- Both the horizontal and vertical diameters are defined as integers.
- Create a round-cornered rectangle by calling the **drawRoundRect() method** and **fillRoundRect() method**, respectively.
- **Both methods require six parameters.**
- The **first four parameters** identify the upper-left corner and lower-right corner of the rectangle.

- The **fifth and sixth parameters** are integers representing the horizontal diameter and vertical diameter of the corners.

### Drawing a Rectangle:

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class RectangleExample extends MIDlet
{
    private Display display;
    private MyCanvas canvas;
    public RectangleExample ()
    {
        display = Display.getDisplay(this);
        canvas = new MyCanvas (this);
    }
    protected void startApp()
    {
        display.setCurrent( canvas );
    }
    protected void pauseApp()
    {
    }
    protected void destroyApp( boolean unconditional )
    {
    }
    public void exitMIDlet()
    {
        destroyApp(true);
        notifyDestroyed();
    }
}
class MyCanvas extends Canvas implements CommandListener
{
    private Command exit;
    private RectangleExample rectangleExample;
    public MyCanvas (RectangleExample rectangleExample)
    {
        this.rectangleExample = rectangleExample;
        exit = new Command("Exit", Command.EXIT, 1);
        addCommand(exit);
        setCommandListener(this);
    }
    protected void paint(Graphics graphics)
    {

```

```
graphics.setColor(255,255,255);
graphics.fillRect(0, 0, getWidth(), getHeight());
graphics.setColor(255,0,0);
graphics.drawRect(2, 2, 20, 20);
graphics.drawRoundRect(20, 20, 60, 60, 15, 45);
}
public void commandAction(Command command, Displayable displayable)
{
    if (command == exit)
    {
        rectangleExample.exitMIDlet();
    }
}
}
```



### Filled rectangle:



```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class FilledRectangleExample extends MIDlet
{
    private Display display;
    private MyCanvas canvas;
    public FilledRectangleExample ()
    {
        display = Display.getDisplay(this);
        canvas = new myCanvas(this);
    }
    protected void startApp()
```



```
{
display.setCurrent( canvas );
}
protected void pauseApp()
{
}
protected void destroyApp( boolean unconditional )
{
}
public void exitMIDlet()
{
destroyApp(true);
notifyDestroyed();
}
}
class MyCanvas extends Canvas implements CommandListener
{
private Command exit;
private FilledRectangleExample filledRectangleExample;
public MyCanvas (FilledRectangleExample filledRectangleExample)
{
this.filledRectangleExample = filledRectangleExample;
exit = new Command("Exit", Command.EXIT, 1);
addCommand(exit);
setCommandListener(this);
}
protected void paint(Graphics graphics)
{
graphics.setColor(255,255,255);
graphics.fillRect(0, 0, getWidth(), getHeight());
graphics.setColor(0, 0, 255);
graphics.fillRect(2, 2, 20, 20);
graphics.fillRoundRect(20, 20, 60, 60, 15, 45);
}
public void commandAction(Command command, Displayable displayable)
{
if (command == exit)
{
filledRectangleExample.exitMIDlet();
}
}
}
```



### Arcs

- An arc is a curved line segment that is used to draw circles, ovals, and other curved images.
- Drawing an arc is a bit tricky because you must define the area of the canvas that will be covered by the arc and the angle used to draw the arc.
- The **first step** in drawing an arc is to decide the area of the canvas that will be covered by the arc. The area is defined as a rectangle rather than the circumference of the arc. Think of this as defining a box within which the angle is drawn.
- **A rectangle is defined by specifying two sets of coordinates.**
- The **first set of coordinates** (x1, y1) set the upper-left corner of the rectangle.
- The **other set of coordinates** (x2, y2) set the lower-right corner of the rectangle.
- Once the rectangle is defined, you must define two angles used to draw the arc.
- An angle is defined in degrees from 0 to 360 degrees.
- The **first angle** is the starting point of the arc, and the **other angle** is the end point of the arc.
- Picture a clock. The 3 o'clock position is 0 degree. Degrees are incremented as you move counterclockwise. The 12 o'clock position is 90 degrees, 9 o'clock is 180 degrees, and 6 o'clock is 270 degrees. Degrees decrement as you move clockwise. Based on the picture of the clock, you must select the angle where the arc begins to be drawn. Likewise, you select the angle where the arc terminates.
- Can draw **two kinds of arcs**
  -  an outlined arc and
  -  a filled arc.
- In an **outlined arc**, only the circumference of the arc is drawn (like a smile).
- In a **filled arc**, the circumference of the arc is drawn, and the area within the center and the circumference is filled with the color of the graphic context used to draw the arc.

- An outlined arc is drawn by calling the **drawArc() method**, and
- the filled arc is drawn by calling the **fillArc() method**.
- **Both methods require six parameters**, all of which are integers.
- The **first two parameters** set the coordinates for the upper-left corner of the rectangle that contains the arc (x1, y1).
- The next **two parameters** are the width and height of that rectangle (x2, y2).
- The **fifth parameter** is the start angle, and
- the **last parameter** is the end angle of the arc.

### Drawing an Arc

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class ArcExample extends MIDlet
{
    private Display display;
    private MyCanvas canvas;
    public ArcExample()
    {
        display = Display.getDisplay(this);
        canvas = new MyCanvas (this);
    }
    protected void startApp()
    {
        display.setCurrent( canvas );
    }
    protected void pauseApp()
    {
    }
    protected void destroyApp( boolean unconditional )
    {
    }
    public void exitMIDlet()
    {
        destroyApp(true);
        notifyDestroyed();
    }
}
class MyCanvas extends Canvas implements CommandListener
{
    private Command exit;
    private ArcExample arcExample;
    public myCanvas (ArcExample arcExample)
    {
        this.arcExample = arcExample;
        exit = new Command("Exit", Command.EXIT, 1);
    }
}
```

```
addCommand(exit);
setCommandListener(this);
}
protected void paint(Graphics graphics)
{
    graphics.setColor(255,255,255);
    graphics.fillRect(0, 0, getWidth(), getHeight());
    graphics.setColor(255,0,0);
    graphics.drawArc(0, 0, getWidth(), getHeight(), 180, 180);
}
public void commandAction(Command command, Displayable displayable)
{
    if (command == exit)
    {
        arcExample.exitMIDlet();
    }
}
}
```



### Filled arc:

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class ArcFilledExample extends MIDlet
{
    private Display display;
    private MyCanvas canvas;
    public ArcFilledExample()
    {

```

```
display = Display.getDisplay(this);
canvas = new MyCanvas (this);
}
protected void startApp()
{
display.setCurrent( canvas );
}
protected void pauseApp()
{
}
protected void destroyApp( boolean unconditional )
{
}
public void exitMIDlet()
{
destroyApp(true);
notifyDestroyed();
}
}
class MyCanvas extends Canvas implements CommandListener
{
private Command exit;
private ArcFilledExample arcFilledExample;
public MyCanvas (ArcFilledExample arcFilledExample)
{
this.arcFilledExample = arcFilledExample;
exit = new Command("Exit", Command.EXIT, 1);
addCommand(exit);
setCommandListener(this);
}
protected void paint(Graphics graphics)
{
graphics.setColor(255,255,255);
graphics.fillRect(0, 0, getWidth(), getHeight());
graphics.setColor(255,0,0);
graphics.fillArc(0, 0, getWidth(), getHeight(), 180, 180);
}
public void commandAction(Command command, Displayable displayable)
{
if (command == exit)
{
arcFilledExample.exitMIDlet();
}
}
}
```



### Text

- Displaying text using the low-level user interface differs from displaying text with the high-level user interface.
- The difference is based on who controls the details used to display the text.
- Using the high-level user interface, text is displayed by calling one of several methods and passing the text as a parameter to those methods. Each method determines how to display the text without requiring any direction from you. This is not the situation when displaying text using the low-level user interface because you control the details of how text is displayed.
- Many developers pay little attention to how text appears on the screen (other than color and position), although displaying text is a complex operation.
- For example, someone must determine the appearance of each letter of the text, the height and width of every character, and the size of the space between characters, among other such details.
- The **font** used to display text determines the appearance of text on the screen.
- There are thousands of fonts, as you have probably seen when you write a document using a word processor. We identify fonts by name, such as Times Roman and Arial.
- A font name actually represents a set of font metrics that determine the pixels necessary to generate alphanumeric characters and symbols on the screen and on a printed page.
- The J2SE specification defines the FontMetrics class that is used to specify every detail of the font; but the J2ME specification does not support the FontMetrics class, and therefore you are limited to the font metrics that you can control from within your MIDlet.
- There **are three font metrics** that are controllable by a MIDlet.
- These are the **font face, the font style, and the font size**.

- Selecting the *font face* is similar to selecting the font name in a word processing document, **although selections are limited to the default system font face, monospace font face, and proportional font face.**
- The **default system is the font face** that the device chooses.
- **Monospace** is a font face in which all characters are the same width.
- **Proportional** is a font face in which the width of a character is determined by the nature of the character.

**For example, the letter W is wider than the letter A, and the letter I has a smaller width than A.**

- There are **four font styles** to choose from, which are identical to styles available in a word processor.
- These are **plain, bold, italic, and underlined.** You can apply multiple font styles to text by using the OR (|) operator.
- **Font sizes are small, medium, and large.**
- The small computing device determines the actual size of the font.
- Unlike a word processor, in which you can set font sizes to 10 points or 12 points, you cannot select specific type sizes. Instead, you must limit your choices to small, medium, or large.
- **Font faces, font styles, and font sizes are associated with font constants** that are used to identify your font request.

Font Constant	Description	Font Constant Value
FACE_SYSTEM	System font face	0
FACE_MONOSPACE	Monospace font face	32
FACE_PROPORTIONAL	Proportional font face	64
STYLE_PLAIN	Plain font style	0
STYLE_BOLD	Bold font style	1
STYLE_ITALIC	Italicized font style	2
STYLE_UNDERLINED	Underlined font style	4
SIZE_SMALL	Small font size	8
SIZE_MEDIUM	Medium font size	0
SIZE_LARGE	Large font size	16



**Table 7-3. Font Constants**

- It is important to understand that your selection of a font is a request and not a directive to the device.
- The device will match your request to available fonts, but there is no guarantee that your request will be fulfilled.
- set a font by calling the **setFont() method**, which is a member of the Graphics class.
- The **setFont() method requires one parameter**, which is an instance of the Font class.
- obtain the instance of the Font class by calling the **getFont() method.**
- The **getFont() method requires three parameters.**
- The **first parameter** is the font face,
- the **second parameter** is the font style, and
- the **last parameter** is the font size.

```
graphics.setFont(Font.getFont(Font.PROPORTIONAL,Font.BOLD | Font.ITALIC,Font.SMALL));
```

- can determine the font face, font style, and font size of an instance of the Font class by calling the **getFace() method, getStyle() method, and the getSize() method, respectively.**
- These methods return an integer that represents the value of the font constant .
- The integer returned by the **getStyle() method** represents the combined style of the **instance of the Font class**, such as **Font.BOLD | Font.ITALIC**.
- can use a series of other methods to query the individual font styles associated with a font.
- These methods are **isPlain(), isBold(), isItalic(), and isUnderlined()**. **Each of these methods returns a Boolean** value that is true if the corresponding style is used in the font, otherwise a false value is returned.

### Aligning Text

- **Aligning text** is probably the trickiest routine that encounter when drawing text on the canvas because need to know measurements of text that is already on the canvas as well as measurements of text being drawn.
- **Text** is drawn within a virtual bounding box, which is an invisible box that defines the boundaries of the text.
- **First specify a position** on the screen by setting coordinates. Let's call them x, y.
- **Next, specify an anchor point** that identifies the relationship of the coordinate to the bounding box.
- Suppose we want the coordinates to be the upper-left corner of the boundary box, then specify the anchor point TOP | LEFT. Or want the coordinates to represent the lower-right corner of the boundary box, then use the BOTTOM | RIGHT anchor point.
- The width and height of the text determine the coordinate of the opposite corner of the boundary box.
- **Anchor points are represented by anchor point constants.**
  -  There are **three horizontal values, LEFT, HCENTER, and RIGHT**, and
  -  **three vertical values, TOP, BASELINE, and BOTTOM.**
- Horizontal and vertical values define the location within the bounding box of the specified coordinate.
- The values are combined to define an anchor point. pick a location on the screen, and if we want that position to be the upper-right of the bounding box, set the anchor point to **GRAPHICS.TOP | GRAPHICS.RIGHT**.



Anchor Point Constant	Description
LEFT	Coordinates represent the left edge of the boundary box.
HCENTER	Coordinates represent the horizontal center of the boundary box.
RIGHT	Coordinates represent the right edge of the boundary box.
TOP	Coordinates represent the top edge of the boundary box.
BASELINE	Coordinates represent the baseline for the text.
BOTTOM	Coordinates represent the bottom edge of the boundary box.

**Table 7-4.** *Anchor Point Constants*

- Other text that abuts the boundary box usually determines text alignment within the boundary box.
- **Text is measured by**
  - ✚ ascent,
  - ✚ descent,
  - ✚ leading,
  - ✚ font height, and
  - ✚ advance.
- **Ascent** is the measurement from the baseline of the text to the top of the highest character in the text.
- **Descent** is from the baseline to the lowest character in the text. Let's examine the following text to identify the ascent and descent: "We work together."
- The **ascent** is the distance between the bottom and the top of the *W* because the *W* is the highest character in the text. The bottom of the *W* is the baseline.
- The **descent** is the distance between the bottom of the *g* and the bottom of the *W*, or the bottom of any of the other characters of the text, because the *g* is the lowest character within the text.
- **Leading** is the distance between the descent and ascent of abutting lines of text.
- The **font height** is the sum of the ascent, leading, and descent.
- And the **advance** is the text length, including spaces between characters.
- can use methods of the `Font` class to determine these measurements.
- Can determine the advance by calling the **`charWidth()` method, the `charsWidth()` method, or the `substringWidth()` method.**
- The **`charWidth()` method** measures the width of one character and **requires one parameter, which is a character.**
- This method returns an integer representing the width of the character in pixels.
- The **`charsWidth()` method** measures a series of characters in a character array.
- **This method requires three parameters.**
- **The first parameter** is the character array.
- **The second parameter** is an integer representing the first character of the series being measured.
- **The last parameter** is an integer representing the length of the series.
- The **`substringWidth()` method** measures a substring of characters within a string and also **requires three parameters.**

- The **first parameter** is the string.
- The **second parameter** is an integer representing the first character of the substring, and
- the **last parameter** is an integer representing the length of the substring.
- The **ascent** is measured by calling the **getBaselinePosition() method**.
- No parameters are required by this method because the method analyzes text already associated with the graphic context used to draw text on the canvas.
- The **getBaselinePosition() method** returns an integer that represents the pixels between the baseline and the top character within the text.
- The font height is measured by calling the **getHeight() method**.
- The **getHeight() method** does not require any parameters and returns an integer representing the pixel measurement of the font height.

### Drawing Methods

- There are **four methods** that you can use to draw text on the canvas.
- choice depends on whether the text is
  - a character,
  - array of characters,
  - a string,
  - or a substring.
- Call the **drawChar() method** if we are drawing one character on the canvas.
- The **drawChar() method requires four parameters**.
- The **first parameter** is the character.
- The **next two parameters** are the x, y coordinates of the upper-left corner of the boundary box.
- And the **last parameter** is the anchor point.
- The **drawChars() method** is used to draw an array of characters or a subset of a character array.
- The **drawChars() method requires six parameters**.
- The **first parameter** is the character array.
- The **second parameter** is an integer representing the offset in the array where the first character to be drawn is located.
- The **third parameter** indicates how many characters are to be drawn starting at the offset specified by the second parameter.
- The **fourth and fifth parameters** are the x, y coordinates of the boundary box's anchor point.
- There are **two methods you can use when drawing a string on the canvas**.
- These are the **drawString() method** and the **drawSubstring() method**.
- The **drawString() method requires four parameters**.
- The **first parameter** is the string.
- The **second and third parameters** are the x, y coordinates that specify the anchor point coordinate.
- And the **last parameter** specifies the part of the boundary box to locate the anchor point.

- The **drawSubstring()** method is very similar in structure to the **drawChars()** method in that the same six parameters are required, except the first parameter is the string that contains the substring that is being drawn on the canvas.

### Drawing Text

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class TextExample extends MIDlet
{
    private Display display;
    private MyCanvas canvas;
    public TextExample ()
    {
        display = Display.getDisplay(this);
        canvas = new MyCanvas (this);
    }
    protected void startApp()
    {
        display.setCurrent(canvas);
    }
    protected void pauseApp()
    {
    }
    protected void destroyApp( boolean unconditional )
    {
    }
    public void exitMIDlet()
    {
        destroyApp(true);
        notifyDestroyed();
    }
}
class MyCanvas extends Canvas implements CommandListener
{
    private Command exit;
    private TextExample textExample;
    public MyCanvas (TextExample textExample)
    {
        this.textExample = textExample;
        exit = new Command("Exit", Command.EXIT, 1);
        addCommand(exit);
        setCommandListener(this);
    }
    protected void paint(Graphics graphics)
    {
        graphics.setColor(255,255,255);
```

```
graphics.fillRect(0, 0, getWidth(), getHeight());
graphics.setColor(255,0,0);
graphics.setFont(Font.getFont(Font.FACE_PROPORTIONAL,
Font.STYLE_BOLD, Font.SIZE_SMALL));
graphics.drawString("Profound statement.", 50, 10,
Graphics.HCENTER|Graphics.BASELINE);
}
public void commandAction(Command command, Displayable displayable)
{
if (command == exit)
{
textExample.exitMIDlet();
}
}
}
```

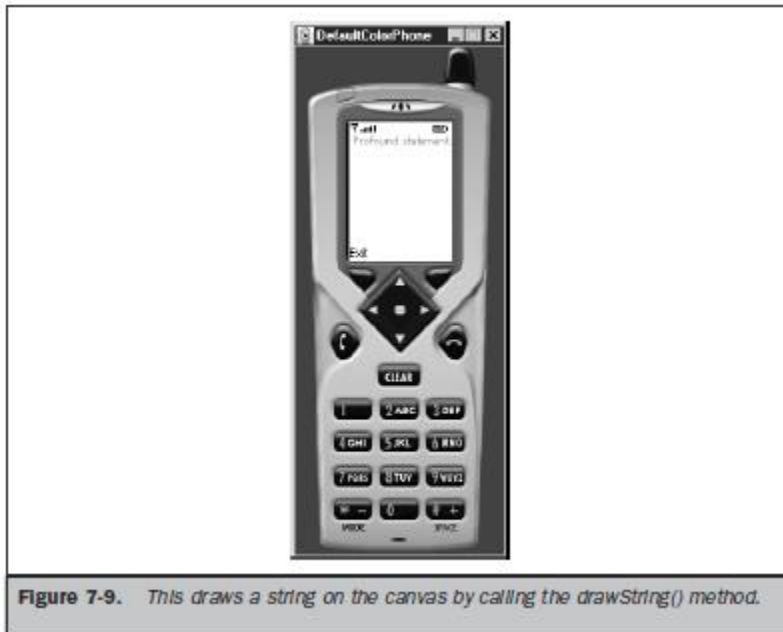


Figure 7-9. This draws a string on the canvas by calling the drawString() method.

### Images:

- An image is an instance of an Image object ,there are **two kinds of images**:
  - a mutable image can be modified by the MIDlet, and
  - an immutable image cannot be modified by the MIDlet.
- create an instance of an Image by calling the **createImage() method**.
- The **createImage() method requires one parameter or two parameters** depending on whether we are drawing a mutable or immutable image.
- **One parameter** is required for the **createImage() method**
- if the instance is used to draw an **immutable image**.
- The **parameter is the file name of the image**, including the full directory path. The following code segment

**Image image = Image.createImage("/myImage.png");**

- **Two parameters are required for the createImage() method**
- if the instance is used to draw a **mutable image**.
- These **parameters define the height and width in pixels of the memory block used to store the mutable image as it is being drawn.**
- The following code segment creates a block of memory 20 pixels high and 10 pixels wide for a total image size of 200 pixels.



**Image tmpImg = Image.createImage(20, 10);**

- use the instance as a parameter to the **drawImage() method**, which draws the image on the canvas.
- The instance of the image is also used to create a mutable image.
- create a mutable image by calling the **getGraphics() method** of the Image class to return an instance of the Graphics class, which is the graphic context used to draw the mutable image.
- Let's say that you want to create a mutable image of a line.
- The following code segment shows how this is done.
- **First**, we must create an instance of the Image class by calling the **createImage() method**.
- **Next**, need an instance of the Graphics class that is created by calling the **getGraphics() method**.
- And then call the **drawLine() method**,

**Image image = Image.createImage(20, 10);**

**Graphics graphic = image.getGraphics();**

**graphic.drawLine(5, 5, 20, 20);**

- The **drawImage() method** is used to draw a mutable or immutable image on the canvas.
- The **drawImage() requires four parameters**.
- The **first parameter** is the instance of the Image class that references the image.
- The **next two parameters** are integers that represent the coordinate used to position the image on the canvas.
- The image is drawn within a virtual boundary box similar to the boundary box used to draw text on the canvas
- The coordinate represents the upper-left corner of the boundary box.
- The **last parameter** is an integer that represents the portion of the image bounding box that is anchored at the specified coordinate.
- The image anchor point is used to finely adjust the location of the image within the boundary box.
- **Two anchor points must be specified in the drawImage() method**
  -  one for the horizontal position and
  -  the other for the vertical position.
- This is nearly identical to the way anchor points are specified in the text drawing methods.

**graphics.drawImage(image, 5, 20, Graphics.HCENTER | Graphics.VCENTER);**

### Drawing an Image

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class MutableImageExample extends MIDlet
{
    private Display display;
    private MyCanvas canvas;
    public MutableImageExample ()
    {
        display = Display.getDisplay(this);
        canvas = new MyCanvas (this);
    }
    protected void startApp()
    {
        display.setCurrent( canvas );
    }
    protected void pauseApp()
    {
    }
    protected void destroyApp( boolean unconditional )
    {
    }
    public void exitMIDlet()
    {
        destroyApp(true);
        notifyDestroyed();
    }
    class MyCanvas extends Canvas implements CommandListener
    {
        private Command exit;
        private MutableImageExample mutableImageExample;
        private Image image = null;
        public myCanvas(MutableImageExample mutableImageExample)
        {
            this.mutableImageExample = mutableImageExample;
            exit = new Command("Exit", Command.EXIT, 1);
            addCommand(exit);
            setCommandListener(this);
        }
        try
        {
            image = Image.createImage(70, 70);
            Graphics graphics = image.getGraphics();
            graphics.setColor(255, 0, 0);
            graphics.fillArc(10, 10, 60, 50, 180, 180);
        }
        catch (Exception error)
        {
        }
    }
}
```

```
Alert alert = new Alert("Failure",
"Creating Image", null, null);
alert.setTimeout(Alert.FOREVER);
display.setCurrent(alert);
}
}
protected void paint(Graphics graphics)
{
if (image != null)
{
graphics.setColor(255,255,255);
graphics.fillRect(0, 0, getWidth(), getHeight());
graphics.drawImage(image, 30, 30, Graphics.VCENTER | Graphics.HCENTER);
}
}
public void commandAction(Command command, Displayable display)
{
if (command == exit)
{
mutableImageExample.exitMIDlet();
}
}
}
```

Example:

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class ImmutableImageExample extends MIDlet
{
private Display display;
private MyCanvas canvas;
public ImmutableImageExample ()
{
display = Display.getDisplay(this);
canvas = new MyCanvas (this);
}
protected void startApp()
{
display.setCurrent( canvas );
}
protected void pauseApp()
{
}
protected void destroyApp( boolean unconditional )
{
}
```

```
public void exitMIDlet()
{
    destroyApp(true);
    notifyDestroyed();
}
class MyCanvas extends Canvas implements CommandListener
{
    private Command exit;
    private ImmutableImageExample immutableImageExample;
    private Image image = null;
    public MyCanvas (ImmutableImageExample immutableImageExample)
    {
        this. immutableImageExample = immutableImageExample;
        exit = new Command("Exit", Command.EXIT, 1);
        addCommand(exit);
        setCommandListener(this);
        try
        {
            image = Image.createImage("/myImage.png");
        }
        catch (Exception error)
        {
            Alert alert = new Alert("Failure",
            "Can't open image file.", null, null);
            alert.setTimeout(Alert.FOREVER);
            display.setCurrent(alert);
        }
    }
    protected void paint(Graphics graphics)
    {
        if (image != null)
        {
            graphics.drawImage(image, 0, 0,
            Graphics.VCENTER | Graphics.HCENTER);
        }
    }
    public void commandAction(Command command, Displayable display)
    {
        if (command == exit)
        {
            immutableImageExample.exitMIDlet();
        }
    }
}
```



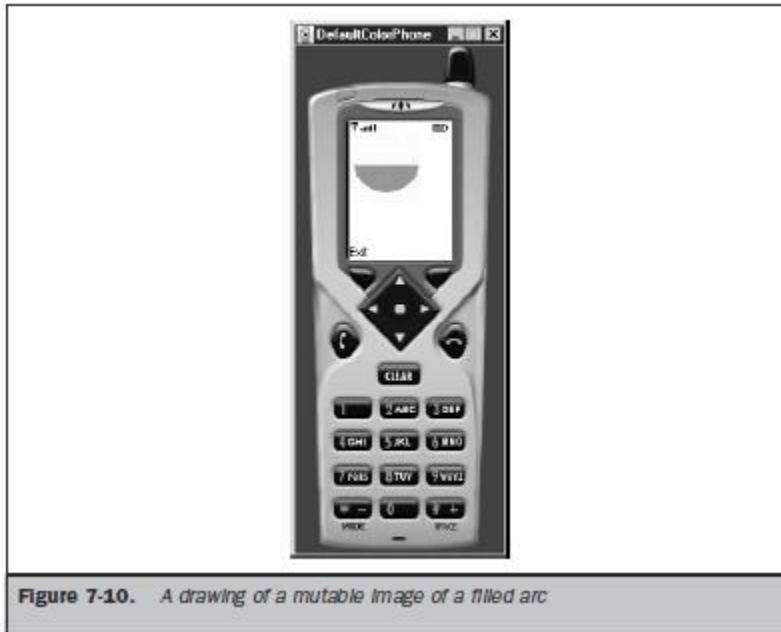


Figure 7-10. A drawing of a mutable image of a filled arc

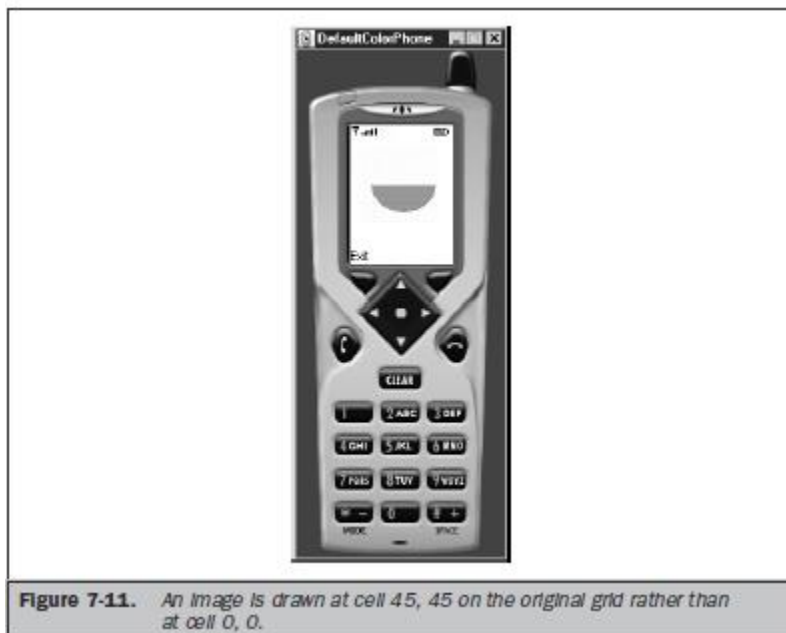
### Repositioning Text and Images

- Each position on a canvas is organized by a row and column grid, where each coordinate identifies a pixel.
- The upper-left corner of the canvas is always coordinate 0, 0 .
- Coordinates are used with methods of a graphic context to identify locations on the canvas for drawing and positioning an image.
- Coordinates are passed explicitly to these methods by providing exact coordinates, such as 5, 10, or implicitly by referencing an offset of an explicit coordinate, such as 5 + 3, 10 + 3.
- In either case, coordinates are based on the 0, 0 coordinate being the upper-left corner of the canvas.
- might need to proportionally shift all text and images to a new location on the canvas.
- There are **two techniques** you can use to make this move.
- **First**, we can change all coordinates of an image to reflect its new position, or you can move the entire grid and let the device adjust all the coordinates based on the new position of the upper-left corner of the grid on the canvas. **This technique, called *translating coordinates***, is a more efficient way of moving text and images than modifying coordinates within your MIDlet. Can translate coordinates by calling the **translate() method** of the Graphics class.
- The **translate() method requires two parameters** that are integers representing the x, y coordinate of the new position of the upper-left corner of the grid.
- can determine the coordinate of the upper-left corner of the canvas by calling the **getTranslateX() method** and **getTranslateY() method** to return the x, y coordinate.

### Translating Coordinates in a MIDlet

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class TranslateCoordinates extends MIDlet
{
    private Display display;
    private MyCanvas canvas;
    public TranslateCoordinates ()
    {
        display = Display.getDisplay(this);
        canvas = new MyCanvas (this);
    }
    protected void startApp()
    {
        display.setCurrent( canvas );
    }
    protected void pauseApp()
    {
    }
    protected void destroyApp( boolean unconditional )
    {
    }
    public void exitMIDlet()
    {
        destroyApp(true);
        notifyDestroyed();
    }
    class MyCanvas extends Canvas implements CommandListener
    {
        private Command exit;
        private TranslateCoordinates translateCoordinates;
        private Image image = null;
        public myCanvas(TranslateCoordinates translateCoordinates)
        {
            this.translateCoordinates = translateCoordinates;
            exit = new Command("Exit", Command.EXIT, 1);
            addCommand(exit);
            setCommandListener(this);
            try
            {
                image = Image.createImage(70, 70);
                Graphics graphics = image.getGraphics();
                graphics.setColor(255,0,0);
                graphics.fillArc(10, 10, 60, 50, 180, 180);
            }
        }
    }
}
```

```
catch (Exception error)
{
    Alert alert = new Alert("Failure",
    "Creating Image", null, null);
    alert.setTimeout(Alert.FOREVER);
    display.setCurrent(alert);
}
}
protected void paint(Graphics graphics)
{
    if (image != null)
    {
        graphics.setColor(255,255,255);
        graphics.fillRect(0, 0, getWidth(), getHeight());
        graphics.translate(45, 45);
        graphics.drawImage(image, 0, 0,
        Graphics.VCENTER | Graphics.HCENTER); }
    }
    public void commandAction(Command command, Displayable display)
    {
        if (command == exit)
        {
            translateCoordinates.exitMIDlet();
        }
    }
}
```



**Figure 7-11.** An image is drawn at cell 45, 45 on the original grid rather than at cell 0, 0.

### Clipping Regions

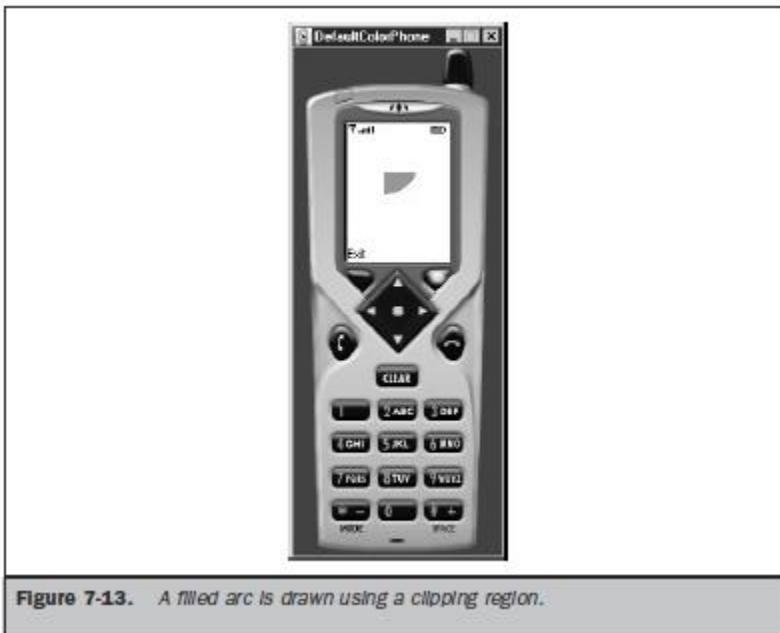
- A **clipping region** is a rectangular piece of an image defined by **two sets of coordinates**.
- The **first set identifies** the upper-left corner of the clipping region, and **the second set** is the width and height of the clipping region.
- Only the portion of the image that appears within the clipping region is drawn on the canvas.
- Other portions of the image still exist within the image but are not drawn.
- The entire canvas is the default clipping region and is used to draw the complete image whenever the **drawImage() method** is called.
- Let's return to the filled arc created and drawn to see how adjusting the clipping region changes the way an image is drawn on the canvas. The image boundary box of the filled arc has the coordinates 10, 10, 60, 60. This means the upper-left corner of the box appears at cell 10, 10 and has a width of 60 and height of 60.
- The clipping region is the entire canvas. We can show a piece of the filled arc by setting the clipping region to 35, 35, 40, 40, which is what happens.
- Only a portion of the filled arc is drawn when the **drawImage() method** is called.
- set the clipping region by calling the **setClip() method** of the Graphics class.
- The **setClip() method requires four parameters**.
- The **first two parameters** are integers representing the upper-left corner coordinates of the clipping region, and
- **the third and fourth parameters** are integers representing the width and height of the clipping region.
- **can also reduce the size** of a clipping region by calling the **clipRect() method**.
- The **clipRect() method** also requires the same **four parameters** as the setClip(), except coordinates passed to the clipRect() method refer to the new clipping region.
- There are **four other methods** that you might find handy when working with a clipping region.
- These are the
  - ✚ **getClipX() method,**
  - ✚ **getClipY() method,**
  - ✚ **getClipHeight() method, and**
  - ✚ **getClipWidth() method.**
- The **getClipX() method and getClipY() method** return upper-left coordinates of the existing clipping region.
- Similarly, the **getClipHeight() method returns the height and getClipWidth() method returns the width of the existing clipping region.**

### Creating a Clipping Region

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class ClippingRegion extends MIDlet
{
    private Display display;
```

```
private MyCanvas canvas;
public ClippingRegion()
{
    display = Display.getDisplay(this);
    canvas = new MyCanvas(this);
}
protected void startApp()
{
    display.setCurrent( canvas );
}
protected void pauseApp()
{
}
protected void destroyApp( boolean unconditional )
{
}
public void exitMIDlet()
{
    destroyApp(true);
    notifyDestroyed();
}
class MyCanvas extends Canvas implements CommandListener
{
    private Command exit;
    private ClippingRegion clippingRegion;
    private Image image = null;
    public MyCanvas (ClippingRegion clippingRegion)
    {
        this.clippingRegion = clippingRegion;
        exit = new Command("Exit", Command.EXIT, 1);
        addCommand(exit);
        setCommandListener(this);
        try
        {
            image = Image.createImage(70, 70);
            Graphics graphics = image.getGraphics();
            graphics.setColor(255,0,0);
            graphics.fillArc(10, 10, 60, 50, 180, 180);
        }
        catch (Exception error)
        {
            Alert alert = new Alert("Failure", "Creating Image",
            null, null);
            alert.setTimeout(Alert.FOREVER);
            display.setCurrent(alert);
        }
    }
}
```

```
}  
protected void paint(Graphics graphics)  
{  
    if (image != null)  
    {  
        graphics.setColor(255,255,255);  
        graphics.fillRect(0, 0, getWidth(), getHeight());  
        graphics.setClip(35, 35, 40, 40);  
        graphics.drawImage(image, 30, 30,  
Graphics.VCENTER | Graphics.HCENTER);  
    }  
}  
public void commandAction(Command command, Displayable display)  
{  
    if (command == exit)  
    {  
        clippingRegion.exitMIDlet();  
    }  
}  
}  
}
```



**Figure 7-13.** A filled arc is drawn using a clipping region.

### Animation:

- **Animation** is the simulation of motion on the screen caused by the timed drawing of a series of related images.
- Each image is referred to as a *cell* in animation terminology; however, we'll use the term *image* instead because a cell also refers to an intersection of the grid used for positioning objects on the canvas.

- Each image in the animation must relate to the image currently displayed and the next image to be displayed.
- Let's say the animation shows a ball bouncing up and down. At least two images are necessary to create this illusion of motion.
- One image is the ball in the air, and the other is the ball on the ground.
- The animation begins by showing the image of the ball in the air followed by the image of the ball on the ground and then back to displaying the ball in the air.
- can enhance the quality of the animation by inserting other images in this sequence.
- one image might be the ball striking the ground followed by another image that shows the ball being flattened by the ground, and then the image of the rounded ball on the ground is displayed before the ball is shown in the air.
- Simulated movement becomes realistic if 30 images (sometimes called frames) are displayed per second.
- This is the display rate used in films and television. However, you can probably use a slower rate for animation on the small computing device. The slower the rate (fewer images per second), the more jerky the movement appears.
- Animation appears to move smoothly if each image in the series represents a small progression toward the final image. Let's return to the bouncing ball. The ball appears to move roughly if there are three images: one image at the top of the roof; another midway to the ground; and the last image of the ball hitting the ground. However, by showing other images of the ball at different stages of falling in a logical sequence, the ball appears to fall smoothly from the roof.
- The **first step** in animation is to carefully lay out the progression of images that you'll need to display in order to create the illusion of movement.
- **Next**, create each image so that each is slightly different from the previous image.
- can create these images as either mutable or immutable.
- Amutable image is created using methods.
- An immutable image is created using graphics software or digital photography.
- Once images are drawn or loaded from a file, display each image by first calling the **createImage() method** to create an instance of the Image class and then calling the **drawImage() method**.
- Timing the display of each image is controlled by a timing loop (sometimes within a while loop) so that the animation recycles to the first image after the last image is displayed.
- **Another way** to incorporate animation in your MIDlet is to create one large image that is a composite of each image in the animation.
- then change the clipping region each time an image is to be displayed. The coordinate of the clipping region changes to the coordinate of the image that is to be displayed.
- This technique draws the image once and then displays portions of the image as required to create the illusion of movement.

### MOBILE APPLICATION DEVELOPMENT

#### 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.
- 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.
- 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).
- 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.
- 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:



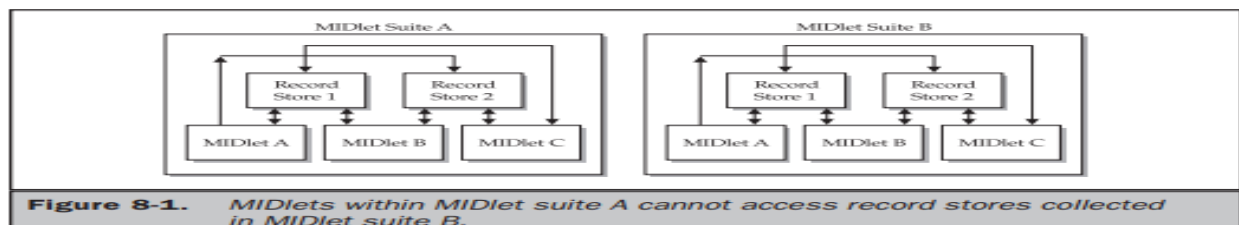
- RMS stores information in a record store. A record store compares to a flat file used for data storage in a traditional file system and to a table of a database.
- A record store contains information referenced by a single name, similar to a flat file and like a table.
- A record store is a collection 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 we 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:

- 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.



### 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.
- can 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()`.**

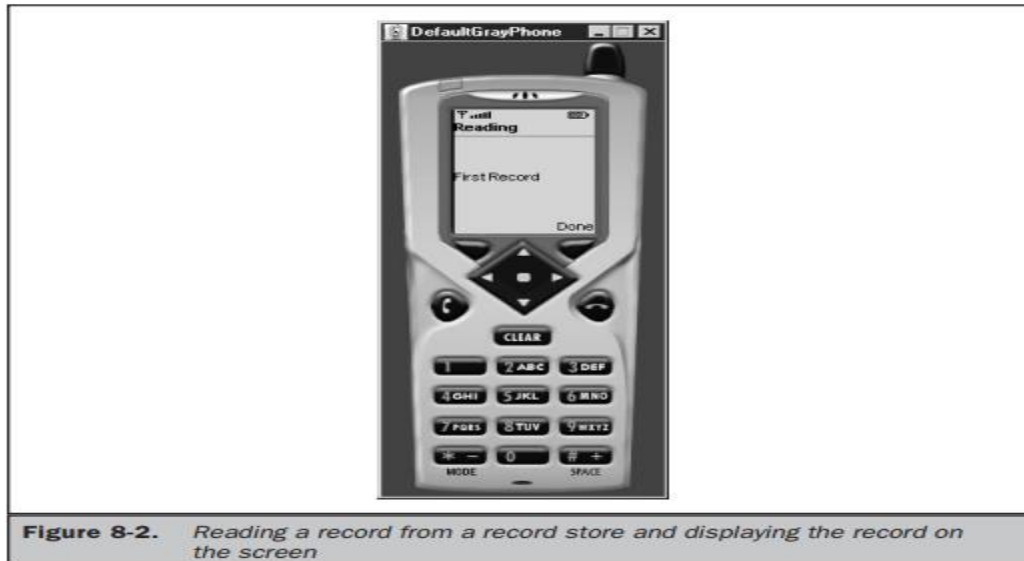
- The second parameter of the `addRecord()` method is usually zero, and the third parameter is the length of the byte array, indicating that the entire byte array should be written to the record store. Information is read from a record store a record at a time and stored in a byte array.
- The byte array 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,
- 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);
} 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", 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); } } } }
```



### Writing and Reading Mixed Data Types:

It is common for records to consist of mixed data types such as string, boolean, and Integer,

ID	Column 1	Column 2	Column 3
1	First Record	15	true
2	Second Record	10	false
3	Third Record	5	true

**Figure 8-3.** A record store typically stores data of multiple data types such as a string, integer, and boolean.

- Once the record is written, the MIDlet reads the context of the record, which is displayed in an alert dialog box.
- The MIDlet creates a record store called myRecordStore after retrieving reference to the display.
- Any errors occurring while the record store is being created are caught by the catch {} block and displayed in an alert dialog box.
- the MIDlet creates an array of bytes called outputRecord followed by the creation of the string, integer, and boolean values of the first record.
- Two streams are used to write the record to the record store. The first stream is a byte array output stream created by calling the ByteArrayOutputStream() constructor.
- The other stream a data output stream used to output the byte array output stream—is created by calling the DataOutputStream() constructor and passing it reference to the byte array outputstream.
- The objective is to write data to a buffer, then write the buffered data to the stream. The data stream is then converted to a byte array. The byte array is then written to the record store.
- The DataOutputStream class has methods that write specific data types to a buffer.
- Three of these methods are used in this example. **These are writeUTF() method, writeBoolean() method, and writeInt() method.**
- The buffered data is placed in the data stream by calling the flush() method.
- The stream is converted to a byte array by calling the toByteArray() method, which returns a reference to the byte array of the stream.
- The ByteArrayOutputStream object's internal store is cleared by calling the reset() method.
- Reading a mixed data type record from a record store is similar to the routine that writes mixed data types. First, you create an instance of the ByteArrayInputStream class.
- The constructor of this class is passed the byte array that was just created. You also create of the DataInputStream class and pass reference to the ByteArrayInputStream class to the DataInputStream class constructor.
- The routine used to read records from a record store must assume that more than one record exists, and therefore you need to include a for loop so the MIDlet continues to read records from a record store until the last record is read.
- The reset() method is called to enable reuse of the ByteArrayInputStream's buffer. The MIDlet then returns to the top of the for loop and evaluates whether or not to read another record from the record store.

```
import javax.microedition.rms.*;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;
public class WriteReadMixedDataTypesExample
extends MIDlet implements CommandListener
{
private Display display;
private Alert alert;
```

```
private Form form;
private Command exit;
private Command start;
private RecordStore recordstore = null;
public WriteReadMixedDataTypesExample ()
{
    display = Display.getDisplay(this);
    exit = new Command("Exit", Command.SCREEN, 1);
    start = new Command("Start", Command.SCREEN, 1);
    form = new Form("Mixed Record");
    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
{
byte[] outputRecord;
String outputString = "First Record";
int outputInteger = 15;
boolean outputBoolean = true;
ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
DataOutputStream outputDataStream =
new DataOutputStream(outputStream);
outputDataStream.writeUTF(outputString);
outputDataStream.writeBoolean(outputBoolean);
outputDataStream.writeInt(outputInteger);
outputDataStream.flush();
outputRecord = outputStream.toByteArray();
recordstore.addRecord(outputRecord, 0, outputRecord.length);
outputStream.reset();
outputStream.close();
outputDataStream.close();
}
catch ( Exception error)
{
alert = new Alert("Error Writing",
error.toString(), null, AlertType.WARNING);
alert.setTimeout(Alert.FOREVER);
display.setCurrent(alert);
}
try
{
String inputString = null;
int inputInteger = 0;
boolean inputBoolean = false;
byte[] byteInputData = new byte[100];
ByteArrayInputStream inputStream = new ByteArrayInputStream(byteInputData);
DataInputStream inputDataStream =
new DataInputStream(inputStream);
for (int x = 1; x <= recordstore.getNumRecords(); x++)
{
recordstore.getRecord(x, byteInputData, 0);
inputString = inputDataStream.readUTF();
inputBoolean = inputDataStream.readBoolean();
inputInteger = inputDataStream.readInt();
inputStream.reset();
}
inputStream.close();
inputDataStream.close();
alert = new Alert("Reading", inputString + " " +
```

```
inputInteger + " " +
inputBoolean, 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)
{
    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 8-4. Displaying mixed data types read from a record store.

### Record Enumeration:

- A record store is more like a flat file than a database management system and therefore lacks many sophisticated features that you find in a database management system.
- Foreexample, you cannot send an SQL query to a record store, nor can you ask a record store to search for keywords or sort records, which is commonly performed by a database management system. can still perform searches and sorts of records in a record store by using the RecordEnumeration interface. obtain a record enumeration by calling the enumerateRecords() method.
- **The enumerateRecords() method requires three parameters.**
- The **first** is the record filter used to exclude records returned from the record store.
- The **second** is reference to the record comparator, which is a method used to compare records returned from the record store.
- The **last parameter** is a boolean value indicating whether or not the enumeration is automatically updated when changes are made to the underlying record store.

**RecordEnumeration recordEnumeration=recordstore.enumerateRecords(null,null,false);**

- The **hasNextElement()** method is called to evaluate whether or not there is another record in the RecordEnumeration.
- A boolean true is returned if another record exists; otherwise, a boolean false is returned.

**while ( recordEnumeration.hasNextElement())**

```
{  
  //do something  
}
```

- **can retrieve a record from the RecordEnumeration using one of two techniques.**
- The **first technique** is designed to read a record that has a single data type such as a string from the RecordEnumeration.
- The **other technique** reads a record that has a compound data type. The next code segment shows how to read a record that has a single data type, which in this case is a string.

- This code segment calls the **nextRecord()** method, which returns a copy of the next record in the RecordEnumeration.
- The record is passed to the constructor of the String class and is assigned to the string variable.  
**String string = new String(recordEnumeration.nextRecord());**  
can move forward or back within the RecordEnumeration by calling either the **nextRecord()** method, which moves to the next record, or the **previousRecord()** method, which moves back one record.
- Both the nextRecord() method and the previousRecord() method return a byte array containing a copy of the record.
- Call the **numRecords()** method to determine the number of records there are in the RecordEnumeration.
- The **numRecords()** method returns an integer representing the total number of records. If the return value is greater than zero, then evaluate whether there is a next record or previous record depending on the desired direction.
- The **hasNextElement()** method is called to determine whether there is a next record, which is illustrated in a previous code segment. Call the **hasPreviousElement()** method to determine whether there is a previous record. Both methods return a boolean value indicating whether or not there is another record.
- The ID of the first record in the RecordEnumeration is zero, and the ID of the last record is nine. You can determine the record ID of the next record by calling the **nextRecordId() method**.
- The **nextRecordId() method** returns an integer representing the ID of the next record. Likewise, you call the **previousRecordId()** method to retrieve the ID of the previous record.
- **There are two ways in which automatic updating is activated or deactivated.**
- The **first way** is when the RecordEnumeration is created.
- the last parameter in the **enumerateRecords()** method is a boolean value that if set to true, causes the RecordEnumeration to update automatically.
- The RecordEnumeration is not changed when the underlying record store changes if the boolean value is false.
- The **other way** to set automatic updating of the RecordEnumeration is by calling the **keepUpdated()** method.
- The **keepUpdated()** method has one parameter, which is a boolean value indicating whether or not the RecordEnumeration is automatically updated. can check the status of the automatic updating feature by calling the **isKeptUpdated()** method. This method returns a boolean value indicating whether or not the RecordEnumeration is automatically updated.
- can manually cause the RecordEnumeration to be rebuilt by calling the **rebuild() method**.
- The **rebuild() method** should be called whenever records in the underlying record store change and the automatic update feature is deactivated.
- The **destroy()** method is called in this example to release resources used by the RecordEnumeration and thereby having the effect of deleting records in the RecordEnumeration.

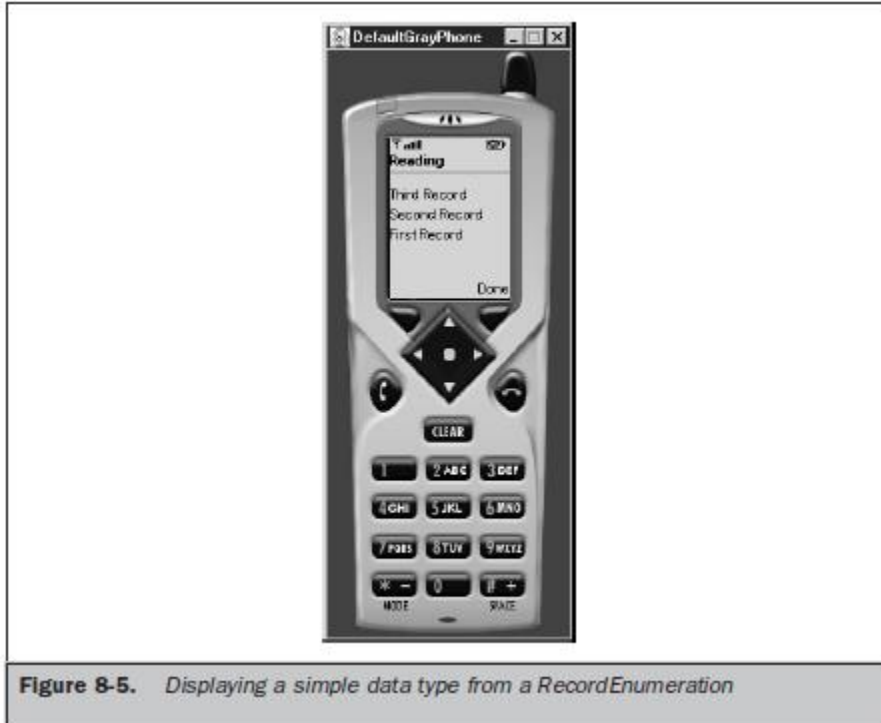
### Reading a Record of a Simple Data Type into a RecordEnumeration:

```
import javax.microedition.rms.*;
```

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;
public class RecordEnumerationExample
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 RecordEnumerationExample ()
    {
        display = Display.getDisplay(this);
        exit = new Command("Exit", Command.SCREEN, 1);
        start = new Command("Start", Command.SCREEN, 1);
        form = new Form("RecordEnumeration");
        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
{
    String outputData[] = {"First Record",
        "Second Record", "Third Record"};
    for (int x = 0; x < 3; x++)
    {
        byte[] byteOutputData = outputData[x].getBytes();
        recordstore.addRecord(byteOutputData,
            0, byteOutputData.length);
    }
}
catch ( Exception error)
{
    alert = new Alert("Error Writing",
        error.toString(), null, AlertType.WARNING);
    alert.setTimeout(Alert.FOREVER);
    display.setCurrent(alert);
}
try
{
    StringBuffer buffer = new StringBuffer();
    recordEnumeration =
        recordstore.enumerateRecords(null, null, false);
    while (recordEnumeration.hasNextElement())
    {
        buffer.append(new String(recordEnumeration.nextRecord()));
        buffer.append("\n");
    }
    alert = new Alert("Reading",
        buffer.toString(), 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)
{
try
{
RecordStore.deleteRecordStore("myRecordStore");
recordEnumeration.destroy();
}
catch (Exception error)
{
alert = new Alert("Error Removing",
error.toString(), null, AlertType.WARNING);
alert.setTimeout(Alert.FOREVER);
display.setCurrent(alert);
}
}
}
}
```



### Reading a Mixed Data Type Record into a RecordEnumeration:

```
import javax.microedition.rms.*;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;

public class MixedRecordEnumerationExample
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 MixedRecordEnumerationExample ()
    {
        display = Display.getDisplay(this);
        exit = new Command("Exit", Command.SCREEN, 1);
        start = new Command("Start", Command.SCREEN, 1);
        form = new Form("Mixed RecordEnumeration");
        form.addCommand(exit);
        form.addCommand(start);
    }
}
```

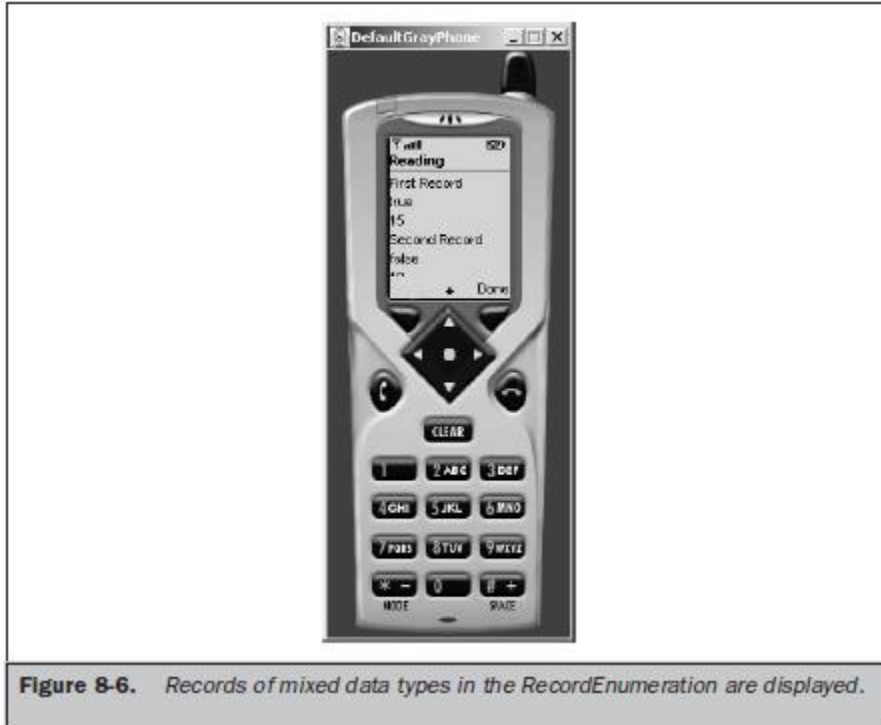


```
form.setCommandListener(this);
}
public void startApp()
{
display.setCurrent(form);
}
public void pauseApp()
{
}
public void destroyApp( boolean unconditional )
{
}
public void commandAction(Command command, Displayable displayable)
{
if (command == exit)
{
destroyApp(true);
notifyDestroyed();
}
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
{
byte[] outputRecord;
String outputString[] = { "First Record",
"Second Record", "Third Record" };
int outputInteger[] = { 15, 10, 5 };
boolean outputBoolean[] = { true, false, true };
ByteArrayOutputStream outputStream =
new ByteArrayOutputStream();
DataOutputStream outputDataStream =
new DataOutputStream(outputStream);
for (int x = 0; x < 3; x++)
{
```

```
outputDataStream.writeUTF(outputString[x]);
outputDataStream.writeBoolean(outputBoolean[x]);
outputDataStream.writeInt(outputInteger[x]);
outputDataStream.flush();
outputRecord = outputStream.toByteArray();
recordstore.addRecord(outputRecord, 0,
outputRecord.length);
}
outputStream.reset();
outputStream.close();
outputDataStream.close();
}
catch ( Exception error)
{
    alert = new Alert("Error Writing",
    error.toString(), null, AlertType.WARNING);
    alert.setTimeout(Alert.FOREVER);
    display.setCurrent(alert);
}
try
{
    StringBuffer buffer = new StringBuffer();
    byte[] byteInputData = new byte[300];
    ByteArrayInputStream inputStream = new ByteArrayInputStream(byteInputData);
    DataInputStream inputDataStream =
    new DataInputStream(inputStream);
    recordEnumeration = recordstore.enumerateRecords(
    null, null, false);
    while (recordEnumeration.hasNextElement())
    {
        recordstore.getRecord(recordEnumeration.nextRecordId(),
        byteInputData, 0);
        buffer.append(inputDataStream.readUTF());
        buffer.append("\n");
        buffer.append(inputDataStream.readBoolean());
        buffer.append("\n");
        buffer.append(inputDataStream.readInt());
        buffer.append("\n");
        alert = new Alert("Reading", buffer.toString(),
        null, AlertType.WARNING);
        alert.setTimeout(Alert.FOREVER);
        display.setCurrent(alert);
    }
    inputStream.close();
}
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)
{
try
{
RecordStore.deleteRecordStore("myRecordStore");
recordEnumeration.destroy();
}
catch (Exception error)
{
alert = new Alert("Error Removing",
error.toString(), null, AlertType.WARNING);
alert.setTimeout(Alert.FOREVER);
display.setCurrent(alert);
}
}
}
}
```



**Figure 8-6.** Records of mixed data types in the RecordEnumeration are displayed.

### Sorting Records

- Records within a RecordEnumeration are sorted by defining a comparator class that is an implementation of the RecordComparator interface.
- Within the comparator class you define a method that has the logic to compare each record to determine whether the record is equal to the current record or should precede or follow the current record within the RecordEnumeration.
- This method, called **compare()**, **requires two parameters, which are two byte arrays that contain the current record and the next record.**
- These byte arrays are then converted to two strings that are compared by using the **compareTo()** method of the String class.
- **The compareTo() method returns an integer that is equal to zero, less than zero, or greater than zero.**
  - **A zero indicates that both strings are the same.**
  - **An integer less than zero indicates that the next record precedes the current record in the RecordEnumeration.**
  - **An integer greater than zero indicates that the next record follows the current record in the RecordEnumeration.**
- Based on the return value of the **compareTo()** method, the **compare()** method returns a predefined comparison value.
- These are
  - RecordComparator.EQUIVALENT,
  - RecordComparator.PRECEDES,

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

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

**Table 8-1.** Comparison Values for the compare() Method

### Sorting Single Data Type Records:

in a RecordEnumeration:

```
import javax.microedition.rms.*;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;

public class SortExample 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;
    private Comparator comparator = null;

    public SortExample ()
    {
        display = Display.getDisplay(this);
        exit = new Command("Exit", Command.SCREEN, 1);
        start = new Command("Start", Command.SCREEN, 1);
        form = new Form("Mixed RecordEnumeration", null);
        form.addCommand(exit);
        form.addCommand(start);
        form.setCommandListener(this);
    }

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

    public void pauseApp()
    {

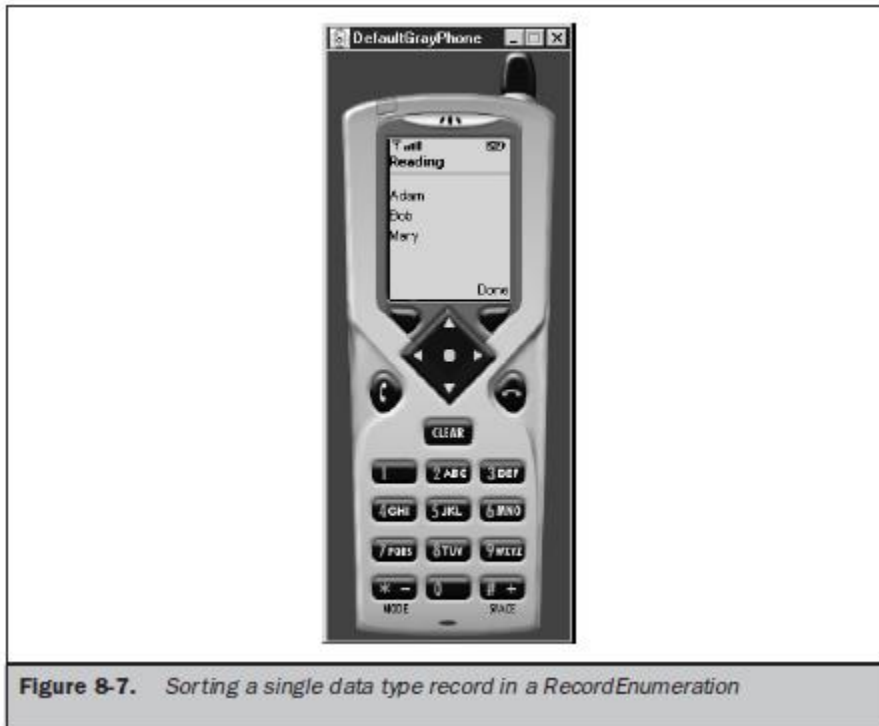
```

```
}
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
{
String outputData[] = {"Mary", "Bob", "Adam"};
for (int x = 0; x < 3; x++)
{
byte[] byteOutputData = outputData[x].getBytes();
recordstore.addRecord(byteOutputData, 0,
byteOutputData.length);
}
}
catch ( Exception error)
{
alert = new Alert("Error Writing",
error.toString(), null, AlertType.WARNING);
alert.setTimeout(Alert.FOREVER);
display.setCurrent(alert);
}
try
{
StringBuffer buffer = new StringBuffer();
Comparator comparator = new Comparator();
```

```
recordEnumeration = recordstore.enumerateRecords(
null, comparator, false);
while (recordEnumeration.hasNextElement())
{
buffer.append(new String(recordEnumeration.nextRecord()));

buffer.append("\n");
}
alert = new Alert("Reading", buffer.toString() ,
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)
{
try
{
RecordStore.deleteRecordStore("myRecordStore");
recordEnumeration.destroy();
}
catch (Exception error)
{
alert = new Alert("Error Removing",
error.toString(), null, AlertType.WARNING);
alert.setTimeout(Alert.FOREVER);
display.setCurrent(alert);
}
}
```

```
}  
}  
  
}  
class Comparator implements RecordComparator  
{  
public int compare(byte[] record1, byte[] record2)  
{  
String string1 = new String(record1),  
string2= new String(record2);  
int comparison = string1.compareTo(string2);  
if (comparison == 0)  
return RecordComparator.EQUIVALENT;  
else if (comparison < 0)  
return RecordComparator.PRECEDES;  
else  
return RecordComparator.FOLLOWS;  
}  
}
```



### Sorting Mixed Data Type Records in a RecordEnumeration:

```
import javax.microedition.rms.*;  
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;  
import java.io.*;
```



```
public class SortMixedRecordDataTypeExample
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;
private Comparator comparator = null;
public SortMixedRecordDataTypeExample ()
{
display = Display.getDisplay(this);
exit = new Command("Exit", Command.SCREEN, 1);
start = new Command("Start", Command.SCREEN, 1);
form = new Form("Mixed RecordEnumeration");
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
{
    byte[] outputRecord;
    String outputString[] = {"Mary", "Bob", "Adam"};
    int outputInteger[] = {15, 10, 5};
    ByteArrayOutputStream outputStream =
        new ByteArrayOutputStream();
    DataOutputStream outputDataStream =
        new DataOutputStream(outputStream);
    for (int x = 0; x < 3; x++)
    {
        outputDataStream.writeUTF(outputString[x]);
        outputDataStream.writeInt(outputInteger[x]);
        outputDataStream.flush();
        outputRecord = outputStream.toByteArray();
        recordstore.addRecord(outputRecord, 0,
            outputRecord.length);
        outputStream.reset();
    }
    outputStream.close();
    outputDataStream.close();

}
catch (Exception error)
{
    alert = new Alert("Error Writing",
        error.toString(), null, AlertType.WARNING);
    alert.setTimeout(Alert.FOREVER);
    display.setCurrent(alert);
}
try
{
    String[] inputString = new String[3];
    int z = 0;
    byte[] byteInputData = new byte[300];
    ByteArrayInputStream inputStream =
        new ByteArrayInputStream(byteInputData);
    DataInputStream inputDataStream =
        new DataInputStream(inputStream);
    StringBuffer buffer = new StringBuffer();
```

```
comparator = new Comparator();
recordEnumeration = recordstore.enumerateRecords(
null, comparator, false);
while (recordEnumeration.hasNextElement())
{
recordstore.getRecord( recordEnumeration.nextRecordId(),
byteInputData, 0);
buffer.append(inputDataStream.readUTF());
buffer.append(inputDataStream.readInt());
buffer.append("\n");
inputDataStream.reset();
}
alert = new Alert("Reading", buffer.toString(), null,
AlertType.WARNING);
alert.setTimeout(Alert.FOREVER);
display.setCurrent(alert);
inputDataStream.close();
inputStream.close();
}
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)
{
try
{
RecordStore.deleteRecordStore("myRecordStore");
comparator.compareClose();
recordEnumeration.destroy();
}
catch (Exception error)
```

```
{
alert = new Alert("Error Removing",
error.toString(), null, AlertType.WARNING);
alert.setTimeout(Alert.FOREVER);
display.setCurrent(alert);
}
}
}
}
}
class Comparator implements RecordComparator
{
private byte[] comparatorInputData = new byte[300];
private ByteArrayInputStream comparatorInputStream = null;
private DataInputStream comparatorInputDataType = null;
public int compare(byte[] record1, byte[] record2)
{
int record1int, record2int;
try
{
int maxlen = Math.max(record1.length, record2.length);
if (maxlen > comparatorInputData.length)
{
comparatorInputData = new byte[maxlen];
}
comparatorInputStream = new ByteArrayInputStream(record1);
comparatorInputDataType =
new DataInputStream(comparatorInputStream);
comparatorInputDataType.readUTF();
record1int = comparatorInputDataType.readInt();
comparatorInputStream = new ByteArrayInputStream(record2);
comparatorInputDataType =
new DataInputStream(comparatorInputStream);
comparatorInputDataType.readUTF();
record2int = comparatorInputDataType.readInt();
if (record1int == record2int)
{
return RecordComparator.EQUIVALENT;
}
else if (record1int < record2int)
{
return RecordComparator.PRECEDES;
}
else
{
return RecordComparator.FOLLOWS;
}
```

```
}  
}  
catch (Exception error)  
{  
return RecordComparator.EQUIVALENT;  
}  
}  
public void compareClose()  
{  
try  
{  
if (comparatorInputStream != null)  
{  
comparatorInputStream.close();  
}  
if (comparatorInputDataType != null)  
{  
comparatorInputDataType.close();  
}  
}  
catch (Exception error)  
{  
}  
}  
}
```



**Figure 8-8.** *Sorting mixed data type records in a RecordEnumeration*

### Searching Records:

- **Searching is referred to as *filtering***, where the filter is defined by the search criteria. Records that match the search criteria are copied into the RecordEnumeration.
- Those not matching the search criteria are filtered from the RecordEnumeration. Searching for a record in a record store is very similar to sorting records in that you define an implementation of an interface.
- In this case the implementation you define filters records contained in a record store rather than sorting records in a RecordEnumeration.
- **The RecordFilter interface is used when searching for a record, must define two methods when defining an implementation of the RecordFilter interface.**
- These are the **matches() method** and the **filterClose() method**.
- The constructor accepts the search criteria as a parameter when your MIDlet creates an instance of the implementation class.
- The **matches()** method contains the logic necessary to determine whether a column fits the search criteria and returns a boolean value indicating whether or not there is a match.
- The **filterClose()** method frees resources used by the implementation of the RecordFilter interface once the search is completed.
- Logic contained in the matches() method reads one or multiple columns from the current record and then applies logical operators to determine whether the record meets the search criteria.
- Can determine the logic used to decide whether or not a record should or should not be included in the RecordEnumeration

### Searching Single Data Type Records:

```
import javax.microedition.rms.*;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;
public class SearchExample 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;
    private Filter filter = null;
    public SearchExample ()
    {
        display = Display.getDisplay(this);
        exit = new Command("Exit", Command.SCREEN, 1);
        start = new Command("Start", Command.SCREEN, 1);
        form = new Form("Mixed RecordEnumeration", null);
```

```
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
{
String outputData[] = {"Mary", "Bob", "Adam"};
for (int x = 0 ; x < 3; x++)
{
byte[] byteOutputData = outputData[x].getBytes();
recordstore.addRecord(byteOutputData, 0,
byteOutputData.length);
}
}
catch ( Exception error)
```

```
{
    alert = new Alert("Error Writing",
        error.toString(), null, AlertType.WARNING);
    alert.setTimeout(Alert.FOREVER);
    display.setCurrent(alert);
}
try
{
    filter = new Filter("Bob");
    recordEnumeration = recordstore.enumerateRecords(
        filter, null, false);
    if (recordEnumeration.numRecords() > 0)
    {
        String string = new String(recordEnumeration.nextRecord());
        alert = new Alert("Reading", string,
            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)
{
    try
    {
        RecordStore.deleteRecordStore("myRecordStore");
        recordEnumeration.destroy();
        filter.filterClose();
    }
}
```



```
catch (Exception error)
{
    alert = new Alert("Error Removing",
        error.toString(), null, AlertType.WARNING);
    alert.setTimeout(Alert.FOREVER);
    display.setCurrent(alert);
}
}
}
}
}
class Filter implements RecordFilter
{
    private String search = null;
    private ByteArrayInputStream inputstream = null;
    private DataInputStream datainputstream = null;
    public Filter(String search)
    {
        this.search = search.toLowerCase();
    }
    public boolean matches(byte[] suspect)
    {
        String string = new String(suspect).toLowerCase();
        if (string!= null && string.indexOf(search) != -1)
            return true;
        else
            return false;
    }
    public void filterClose()
    {
        try
        {
            if (inputstream != null)
            {
                inputstream.close();
            }
            if (datainputstream != null)
            {
                datainputstream.close();
            }
        }
        catch (Exception error)
        {
        }
    }
}
```



**Figure 8-9.** Searching a single data type record in a RecordEnumeration

### Searching Mixed Data Type Records:

```
import javax.microedition.rms.*;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;

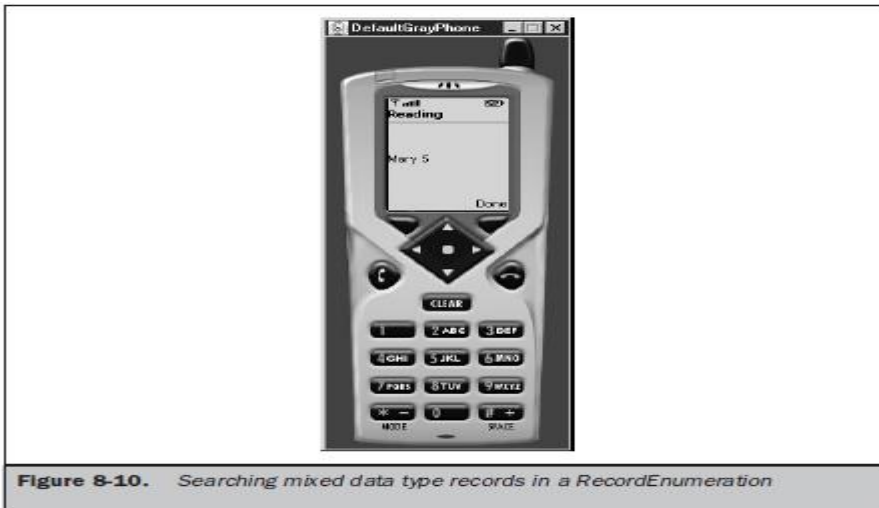
public class SearchMixedRecordDataTypeExample
extends MIDlet implements CommandListener
{
    private Display display;
    private Alert alert;
    private Form form;
    private Command exit;
    private Command start;
    private RecordStore recordstore = null;
    private RecordEnumeration recordEnumeration = null;
    private Filter filter = null;
    public SearchMixedRecordDataTypeExample ()
    {
        display = Display.getDisplay(this);
        exit = new Command("Exit", Command.SCREEN, 1);
        start = new Command("Start", Command.SCREEN, 1);
        form = new Form("Mixed RecordEnumeration");
        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
{
byte[] outputRecord;
String outputString[] = {"Adam", "Bob", "Mary"};
int outputInteger[] = { 15, 10, 5 };
ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
DataOutputStream outputDataStream =
new DataOutputStream(outputStream);
for (int x = 0; x < 3; x++)
{
outputDataStream.writeUTF(outputString[x]);
outputDataStream.writeInt(outputInteger[x]);
outputDataStream.flush();
}
```

```
outputRecord = outputStream.toByteArray();
recordstore.addRecord(outputRecord, 0, outputRecord.length);
outputStream.reset();
}
outputStream.close();
outputDataStream.close();
}
catch ( Exception error)
{
    alert = new Alert("Error Writing",
        error.toString(), null, AlertType.WARNING);
    alert.setTimeout(Alert.FOREVER);
    display.setCurrent(alert);
}
try
{
    String inputString;
    byte[] byteInputData = new byte[300];
    ByteArrayInputStream inputStream =
        new ByteArrayInputStream(byteInputData);
    DataInputStream inputDataStream =
        new DataInputStream(inputStream);
    if (recordstore.getNumRecords() > 0)
    {
        filter = new Filter("Mary");
        recordEnumeration = recordstore.enumerateRecords(
            filter, null, false);
        while (recordEnumeration.hasNextElement())
        {
            recordstore.getRecord(recordEnumeration.nextRecordId(),
                byteInputData, 0);
            inputString = inputDataStream.readUTF() +
                " " + inputDataStream.readInt();
            alert = new Alert("Reading", inputString,
                null, AlertType.WARNING);
            alert.setTimeout(Alert.FOREVER);
            display.setCurrent(alert);
        }
    }
    inputStream.close();
}
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)
{
try
{
RecordStore.deleteRecordStore("myRecordStore");
filter.filterClose();
recordEnumeration.destroy();
}
catch (Exception error)
{
alert = new Alert("Error Removing",
error.toString(), null, AlertType.WARNING);
alert.setTimeout(Alert.FOREVER);
display.setCurrent(alert);
}
}
}
}
}
}
class Filter implements RecordFilter
{
private String search = null;
private ByteArrayInputStream inputstream = null;
private DataInputStream datainputstream = null;
public Filter(String searchcriteria)
{
search = searchcriteria;
}
public boolean matches(byte[] suspect)
{
String string = null;
try
{
```

```
inputstream = new ByteArrayInputStream(suspect);
datainputstream = new DataInputStream(inputstream);
string = datainputstream.readUTF();
}
catch (Exception error)
{
return false;
}
if (string!= null && string.indexOf(search) != -1)
return true;
else
return false;
}
public void filterClose()
{
try
{
if (inputstream != null)
{
inputstream.close();
}
if (datainputstream != null)
{
datainputstream.close();
}
}
catch (Exception error)
{
}
}
```



**Figure 8-10.** Searching mixed data type records in a RecordEnumeration

### RecordListener:

- Records in a RecordEnumeration might be a subset of records if filtering has been applied.
- It is possible that the RecordEnumeration becomes out of sync with the record store if the record store is accessible to other MIDlets.
- Can synchronize the RecordEnumeration and the record store by setting the third parameter of the enumerateRecords() method to true.
- However, besides synchronizing the RecordEnumeration and the record store, there are occasions when you'll want other events to occur whenever a record changes in the record store.
- **The instance of the RecordListener interface is notified whenever one of three changes is made to the record store.**
- **These are when a record is added, modified, or deleted from the record store.**
- **Definition of the RecordListener interface must define three methods:**
  - recordAdded(),
  - recordChanged(),
  - recordDeleted().
- **All three methods require two parameters.**
- The **first parameter** is reference to the record store that has changed, and
- The **second** is an integer indicating the record ID that was added, modified, or removed from the record store.

```
recordstore.addRecordListener( new MyRecordListener());
class MyRecordListener implements RecordListener
{
    public void recordAdded(RecordStore recordstore, int recordid)
    {
        try
        {
            //do something
        }
        catch (Exception error)
        {
            //do something
        }
    }
    public void recordDeleted(RecordStore recordstore, int recordid)
    {
        try
        {
            //do something
        }
        catch (Exception error)
        {
            //do something
        }
    }
}
```

```
}  
public void recordChanged(RecordStore recordstore, int recordid)  
{  
    try  
    {  
        //do something  
    }  
    catch (Exception error)  
    {  
        //do something  
    }  
}
```

### CONCEPT OF DATABASE

- J2ME application saves and retrieves data using the Record Management System (RMS), Applications running on both Connected Limited Device Configuration (CLDC) and Connected Device Configuration (CDC) devices use RMS for local data management.
- J2ME applications that run on CDC devices are also capable of utilizing a relational database management system (DBMS), which provides industrial-strength database management service to the application.
- The DBMS is typically located on a server connected to the device over a network, although some CDC devices might have the DBMS stored locally on a hard drive.
- **CLDC** devices usually have between 160KB and 512KB of available memory and are battery powered.
- They also use an inconsistent, small- bandwidth network wireless connection and may not have a user interface.
- **CLDC** devices use the KJava 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.
- **CDC** devices use a 32-bit architecture, have at least 2MB 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.
- A database is a collection of data managed by a DBMS. Many corporations use one of several commercially available DBMSs, such as Oracle, DB2, Sybase, and Microsoft Access (used for small data collections).
- A CDC-based J2ME application interacts with commercial DBMSs by using a combination of Java data objects that are defined in the Java Database Connection (JDBC) specification and by using the Structured Query Language (SQL).
- The JDBC interface forms a communications link with a DBMS, while SQL is the language used to construct the message (called a *query*) that is sent to the DBMS to request, update, delete, and otherwise manipulate data in the DBMS.



### DATA:

- The **term data** is commonly confused with the term *information*. Although these terms have a similar meaning in the vernacular, they are different when related to a DBMS.
- Information consists of one or more words that collectively infer a meaning, such as a person's address. Data refers to an atomic unit that is stored in a DBMS and is sometimes reassembled into information.

### DATABASES:

- A database in the purest sense is a collection of data. DBMSs use proprietary and public domain algorithms to assure fast and secure interaction with data stored in the database.
- Most DBMSs adhere to a widely accepted relational database model.
- A *database model* is a description of how data is organized in a database.
- In a *relational database model*, data is grouped into tables using a technique called normalization,
- Once a database and at least one table are created, a J2ME application can send SQL statements to the DBMS to perform the following:
  - Save data
  - Retrieve data
  - Update data
  - Manipulate data
  - Delete data

### TABLES:

- A **table** is the component of a database that contains data in the form of rows and columns, very similar to a spreadsheet.
- A **row** contains related data such as clients' names and addresses.
- A **column** contains like data such as clients' first names. Each column is identified by a unique name, called a column name, that describes the data contained in the column.
- An *attribute* describes the characteristic of data that can be stored in the column. Attributes include size, data type, and format.
  - Database name,
  - table name,
  - column name,
  - column attributes, and other information

that describe database components are known as *metadata*.

- **Metadata** is data about data. **Metadata** is used by J2ME applications to identify database components without needing to know details of a column, table, or the database.

### DATABASE SCHEMA:

- A *database schema* is a document that defines all components of a database, such as tables, columns, and indexes.
- A **database schema** also shows relationships between tables; the relationships are used to join rows of two tables,
- **To create a database schema, you must perform six steps:**

- Identify information used in the existing system or legacy system that is being replaced by the J2ME application.
- Decompose this information into data.
- Define data.
- Normalize data into logical groups.
- Create primary and foreign keys.
- Group data together into logical groups.

### Identifying Information:

- The initial step in defining a database schema is to **identify all information** used by the system that is being converted to J2ME technology.
- Information is associated with objects—also known as *entities*—of the system.
- Don't confuse an **entity attribute with data attributes because an entity attribute can be different from data attributes**.
- An **entity attribute** provides general information about an entity
- A **data attribute** provides information about data that is used by the entity.
- Identifying attributes is intuitive most times because an attribute is information commonly used to describe an entity.
- The bestway to identify attributes of an entity is by analyzing instances of the entity.
- An entity is like an empty order form and an instance is an order form that contains order information.
- Once attributes are identified, you must describe the characteristics of each attribute  
Here are common characteristics found in many attributes:
  - ❖ **Attribute name** The name of the attribute uniquely distinguishes the attribute from other attributes of the same entity. "First name" is an attribute name. Duplicate attribute names within the same entity are prohibited. However, two entities can use the same attribute name. That is, the customer entity and the sales representative entity can both have an attribute called first name.
  - ❖ **Attribute type** An attribute type is nearly identical to the data type of a column in a table. Common attribute types include numeric, character, alphanumeric, date, time, Boolean, integer, float, and double, among other attribute types. However, unlike a data type for a column, you do not have to use precise terminology to describe an attribute type.
  - ❖ **Attribute size** The attribute size describes the number of characters used to store values of the attribute. This is similar to the size of a column in a table.
  - ❖ **Attribute range** An attribute range contains minimum and maximum values that can be assigned to an attribute. For example, the value of the "total amount" attribute of an order entity is likely to be greater than zero and less than 10,000, assuming that no order has ever been received that had a total amount of more than 9,999. This range is then used to throw an error should an order be received with a total amount outside this range.
  - ❖ **Attribute default value** An attribute default value is the value that is automatically assigned to the attribute if the attribute isn't assigned a value by the J2ME application. For example, the J2ME application uses the default

system date for the date of an order if a sales representative fails to date the order. The system date is the attribute default value.

- ❖ **Acceptable values** An acceptable value for an attribute is one of a set of values established by the business unit and includes zip codes, country codes, methods of delivery, and simply “yes” or “no.”
- ❖ **Required value** An attribute may require a value before the attribute is saved to a table. For example, an order entity has an order number attribute that must be assigned an order number.
- ❖ **Attribute format** The attribute format consists of the way an attribute appears in the existing system, such as the format of data.
- ❖ **Attribute source** The attribute source identifies the origin of the attribute value. Common sources are from data entry and J2ME applications (such as using the system date as the value of the attribute).
- ❖ **Comments** A comment is free-form text used to describe an attribute.

### Decomposing Attributes to Data:

- Once attributes of entities are identified, they must be reduced to data elements. This process is called *decomposing*.
- In some systems it makes sense to further decompose the customer number attribute, and in other systems no further decomposition of the customer number is necessary.
- The nature of the system will determine whether or not additional decomposition is required for an attribute.

### How to Decompose Attributes

- The process of decomposing attributes begins by analyzing the list of entities and their attributes.
- The list of attributes represents all the information used by the existing system.
- The objective is to reduce each attribute to a list of data that represents the atomic level of the attribute. Here's how to do this:
  1. Look at each attribute and ask yourself if the attribute is atomic.
  2. If the attribute isn't atomic, it must be further decomposed. Create a list of data derived from the attribute, as illustrated in Figure 9-5.
  3. If the attribute is atomic, no further decomposition is necessary for that attribute.
  4. Place the name of the attribute on the data list.
  5. Review the data list developed in step 2 and repeat the decomposition process until all attributes are atomic.

### Defining Data

- Decomposing attributes results in the identification of data elements used by the existing system. Each data element must be defined using techniques similar to those used to describe an attribute. Here are common ways to define a data element:
  - **Data name** The unique name given to the data element, which should reflect the kind of data (see “The Art of Choosing a Name”).
  - **Data type** A data type describes the kind of values associated with the data (see the next section, “Data Types”).
  - **Data size** The size of text data is the maximum number of characters required to represent values of the data. The size of numeric data is usually either the number of digits or the number of bytes for binary representation (for example, smallint in DB2 is 2 bytes).

## Data Types

- the characteristics of data associated with a data element. For example, a street address is likely to be an alphanumeric data type because a street address has a mixture of characters and numbers.
- It is important to use care when selecting the data type of a data element at this stage in the analysis because the data type that you choose typically becomes the data type of the column in the table that contains the data.
- Where possible, limit the choice of data types to those that are common to commercial DBMSs. Not all DBMSs use the same data types; some enhance the standard data type offering. However, if you are unsure of the data type to describe a data element, describe

## The Art of Choosing a Name:

- Picking a name for a data element might seem intuitive, but can easily become tricky when you realize the name must describe the data, it must be unique, and it may have size and character limitations, depending on the DBMS.
- The nature of the data provides a hint to the data name, such as “first name” being used as the name for data associated with a person’s first name
- A data name can be abbreviated using components of the name.
- This means that cust is the abbreviation for customer and should be used in other data of the customer entity (for example, cust bus phone).
- Readability can be enhanced by capitalizing the first letter of each word in the name, or separating these words with hyphens or underlines as shown here:
  - CustHomePhone
  - cust-home-Phone
  - cust\_home\_phone

the type of data in your own words and include an example of data values that are associated with the data element. can refine your choice to available data types in the DBMS when you create the table used to store the data element.

- Many commercially available DBMSs have adopted a common set of data types based on the SQL set of data types, which you’ll learn about in detail in the next chapter.

### Some of these are listed here:

- **Character, also referred to as text** Stores alphabetical characters and punctuation
- **Alpha** Stores only alphabetical characters
- **Alphanumeric** Stores alphabetical characters, punctuation, and numbers
- **Numeric** Stores numbers only
- **Date/Time** Stores dates and time values
- **Logical (Boolean)** Stores one of two values: true or false, 0 or 1, or yes or no
- **LOB (large object)** Stores large text fields, images, and other binary data

## Normalizing Data

- **Normalization** is the process of organizing data elements into related groups to minimize redundant data and to assure data integrity.
- Redundant data elements occur naturally since multiple entities have the same data elements For transactional databases, redundant data makes a database complex, inefficient, and exposes the database to problems referred to as *anomalies* when the DBMS maintains the database.

- Anomalies occur whenever new data is inserted into the database and when existing data is either modified or deleted, and can lead to a breach in referential integrity of the .It minimizes the number of joins and allows data to be summarized into logical groups.
- Errors caused by redundant data are greatly reduced and possibly eliminated by applying the normalization process to the list of data elements that describe all the entities in a system. This is called normalizing the logical data model of a system.
- The normalization process consists of applying a series of rules called *normal forms* to the list of data elements to:
  - Remove redundant data elements.
  - Reorganize data elements into groups.
  - Define one data element of the group (called a *primary key*) to uniquely identify the group. Often, two or more data elements make up the primary key, which is referred to as a *composite key*.
  - Make other data elements of the group (called *non-key* data elements) functionally dependent on the primary key.
  - Relate one group to another using the primary key.For example, a customer number is the primary key of a group that contains customer information. Other data contained in the group such as the customer first name and last name are referred to as non-key data elements. Non-key data elements are functionally dependent on the primary key.

### The Normalization Process

- First normal form (1NF) requires that information is atomic, as discussed previously in this chapter.
- Second normal form (2NF) requires data to be in the first normal form. In addition, data elements are organized into groups eliminating redundant data. Each group contains a primary key and non-key data, and non-key data must be functionally dependent on a primary key.
- Third normal form (3NF) requires that data elements be in the second normal form, and non-key data must not contain transitive dependencies.

### Grouping Data

- A common way to organize data elements into groups is to first assemble a list of all data elements.
- When this is done, will notice that some data elements are duplicated because they are used by more than one entity.
- Duplicate data elements must be removed from the list. Although this is an intuitive process, you must be careful because not all data elements with similar sounding names are duplicates.

### Creating Primary Keys

- A primary key is a data element that uniquely identifies a row of data elements within a group.
- The data selected to become the primary key may or may not exist in the data list generated as the result of analyzing entities.
- Sometimes a data element, such as an order number, is used as the primary key. Other times, the DBMS can be requested to automatically generate a primary key whenever a column in the group isn't suitable to be designated the primary key.
- Let's use a customer entity as an illustration. In the purest sense, a customer has a name and address as attributes. These attributes decompose to first name, last name, Intuitively, the customer first name and last name seem to uniquely identify a customer, but upon closer analysis you'll see that more than one customer might have the same first name and last name. Likewise, two people at the same address might have the same name, although it's somewhat unlikely. If neither a single data element nor a combination of data elements uniquely identify a row, then you must create another data element to serve as the primary key of the table, which is what is required in the previous example. Alternatively, you can request the DBMS to generate a primary key automatically.
- Nearly all commercial DBMSs can generate primary keys to make the database thread safe and reliable. In contrast, a J2ME application that generates a key must contain the logic to be sure that none of the components running on different servers accidentally generate the same key.

### Functional Dependency

- Afunctional dependency occurs when data depends on other data, such as when non-key data is dependent on a primary key. This means that all non-key data has a functional dependency on the primary key within its group.

### Transitive Dependencies

- A transitive dependency is a functional dependency between two or more non-key data elements. This is an elusive concept at first, but an example will clearly illustrate transitive dependency.
- The first grouping in Order entity with the two data elements Sales Rep Number and Sales Region, which is the region to which the sales representative is assigned. Salesperson and region have a transitive dependency.
- The region is functionally dependent on the salesperson, and the salesperson is functionally dependent on the order number. Both salesperson and region are non-keydata and are therefore functionally mutually dependent. The problem lies with the fact that a salesperson cannot be relocated to a different region without having to modify the region data element in the order information group. Therefore, data elements must be regrouped to conform to the third normal form and eliminate transitive dependency.
- The second grouping in the regrouping of the Order entity to address the transitive dependency problem. Notice that a new group is formed that contains the Sales Rep

Number and the Sales Region. The Sales Rep Number and the Sales Region can be used as a composite key.

- Identifying transitive dependencies is tricky. You have to carefully analyze the data elements once the list of data elements is in the second normal form to spot transitive dependencies.

### Foreign Keys

- A foreign key is a primary key of another group used to draw a relationship between two groups of data elements .
- Relationships between twogroups are made using the value of a foreign key and not necessarily the name of a foreign key.

### Referential Integrity

- The success of a relational database is based on the existence of primary keys and foreign keys of data groups to create relationships among groups.
- The existence of this relationship is called referential integrity and is illustrated in Figure 9-10.
- Referential integrity is enforced by imposing database constraints. This means the DBMS assures referential integrity by preventing primary and foreign keys from being modified or deleted.
- Likewise, database constraints prevent new rows from being inserted without maintaining referential integrity.

### The Art of Indexing

- An index is used to quickly locate information in a table, similar to the way information is located in a book.
- However, instead of page numbers, an index references the row number in the table that contains the search criteria.
- **Conceptually, an index is a table that has two columns. One column is the key to the index, and the other column is the number of the row in the corresponding table that contains the value of the key.**

### An Index in Motion

- Unlike an index of a book, a table can be associated with multiple indexes, each of which contains a different key.
- A J2ME application sends the DBMS a query that contains search criteria for information required by the component. Instead of searching the table that contains the search criteria, the DBMS compares the search criteria to keys of indexes, looking for an index to use in the search.



- The request for information (called a query) is sent to the DBMS along with the customer number. The DBMS recognizes that the request contains a customer number, then searches a catalog of indexes for an index that uses customer number as its key.
- The DBMS always uses an index, if one exists, to locate search criteria. If an index doesn't match the search criteria, the DBMS either creates a temporary index as part of the search or sequentially searches the table. The method used with your DBMS depends on how the manufacturer designed the DBMS.
- Once an index is located or a temporary index is created, the DBMS compares the search criteria contained in the request to the index key.
- When a match is found, the DBMS notes the row number in the second column of the index, then opens the associated table and moves directly to the row that corresponds to the row number.
- The DBMS then selects columns from the row that contains customer information requested by the J2ME application.
- 

### Drawbacks of Using an Index

- An index offers an unparalleled advantage for finding information in a table quickly.
- However, there is a drawback when too many indexes are used with one table.
- An unacceptable delay can occur whenever a row is inserted into or deleted from the table.
- Once an index is built, the DBMS is responsible for automatically maintaining the index whenever a row is inserted or deleted from the table.
- **Performance degradation** can be minimized by using a publisher-consumer database design for applications where rows are frequently inserted, deleted, or modified, as in an order processing system.
- The **publisher-consumer database design consists of two or more databases** that contain the same information.
- **One database, called the *publisher***, receives requests from the J2ME application to insert a new row, modify, or delete an existing row. J2ME applications don't use the publisher to retrieve information, therefore the publisher database isn't indexed.
- **A *consumer database*** receives instructions to insert a new row, modify, or delete an existing row from the publisher. However, the consumer receives requests for information from a J2ME application. Therefore, the consumer database is indexed.
- **There are two major benefits to this design.**
- **First**, a bottleneck is avoided when many requests are received by the DBMS to insert new information or update existing information. This is because the publisher has two tasks to perform: insert or modify the database, then pass along those changes to the consumer.
- **The other benefit** is that the database becomes scalable. This means that the database can be adjusted to handle an increased volume of requests.

### Clustered Keys



- A **clustered key** is an index key whose value represents data contained in multiple columns of the corresponding table.
- Although there is only one column in the index to store the index key, that value can be a combination of data from multiple columns of the table.
- **Customer name is composed of two data elements: the customer first name and the customer last name. Since there is only one column for the index key, the customer first name and the customer last name are concatenated into one value.**
- That is, the DBMS takes one data element and places it behind the other data element to create a new data element that becomes the index key.
- Notice there isn't a space between the first and last name. This is because concatenating places the character of the second value immediately following the last character of the first value.
- Clustered keys add overhead to a DBMS because columns are delimited, and the delimiter must be escaped if it appears in the data.
- **Use concatenation of data elements to:**
  - **Create an index key that uses two or more columns to uniquely identify rows in a table**
  - **Facilitate searching for values of multiple columns, such as a customer name, using one index**

### Derived Keys

- A **derived key** consists of a value that represents part of the value of a column rather than the entire value of the column.
- **Let's say that an order number comprises three components: the first component represents the sales region where the order was placed, the second is the sales representative's number, and the third is a unique number that identifies the order.**
- Although the order number appears in one column of the order table, a component of the order number can be used as an index key.

### Selective Rows

- Typically, all the rows in a table are represented in an index associated with that table. However, an index can be created that references a subset of rows in a table.
- The subset is determined when the index is created.
- Suppose a J2ME application is used only to search for orders within a specified region. There will never be an occasion for the component to search other than in the region.
- Indexes used for searches by the component can be limited to only rows of the table that contain orders placed within the region as identified by the first component of the order number.
- This means that the index does not contain any references to rows in the order table that are outside the specified region.

- A performance benefit is realized by using an index that contains a subset of all the rows of the associated table.
- This is because rows that will never be searched are excluded from the index, thereby reducing the number of rows of the index that must be searched by the DBMS.
- The boost in performance, however, is only realized in databases that contain huge amounts of rows. Little if any increase in performance is realized by creating a subset of rows in a typical database because many DBMSs have been optimized to search volumes of data without having to use a subset of rows in an index.

### **Exact Matches and Partial Matches**

- A DBMS can be instructed to use an index to find an exact or partial match to the search criteria. By default, the DBMS searches for exact matches whenever a query is received from a J2ME application.
- An exact match requires that all the characters of the search criteria match the index key.
- If a single character is mismatched, the DBMS does not return data in the corresponding row of the table. A partial match requires that some—not all of the characters of the index key match the search criteria. That is, if the first character of the search criteria and the index key are the same and the remaining characters are different, the DBMS still considers it a match and returns data in the corresponding row of the table.
- Exact matches are used whenever a particular value, and only that value, is required, such as a specific customer number.
- Partial matches are used whenever someone is unsure of the exact value.
- The most significant value in the customer name is the customer last name because there are more people with the same first name than the same last name.

### **Searching for Phonetic Matches:**

- Some DBMSs feature phonetic searches in which the DBMS returns rows containing index keys that “sound” like the search value.
- This means that the DBMS stores both exact spelling and phonic spelling of the index key. Phonetic searches are a valuable feature to look for in a DBMS, especially for use with customer service databases.
- Although a phonetic search won’t guarantee better communication between the customer service representative and the customer, it does give the customer service representative a tool for locating information necessary to respond properly to a customer.
- Phonetic searches are made possible by an algorithm built into the DBMS. The phonetic algorithm used by the DBMS defines each index key phonetically. Likewise, the DBMS converts the search criteria into its phonetic spelling before comparing the phonetic spelling of the search criteria to the phonetic spelling of the index key.

### MOBILE APPLICATION DEVELOPMENT

#### UNIT-V

#### JDBC and Embedded SQL

- A ConnectedDeviceConfiguration(CDC)device-basedJ2MEapplicationinteracts with a database management system (DBMS) using Java data objects and SQL statements that are embedded in the J2ME application and executed by the Java dataobjects.

##### Model Programs:

- Many programming styles can be used to write the data-access portion of a J2ME component.
- **Two programming styles—referred to as ModelA and ModelB.**
- TheModelA program style is used to execute SQL requests that use **the execute() or executeUpdate()** methods and don'treturn a ResultSet.
- TheModelB program style is used to execute SQL requests that use the executeQuery() method, which returns a ResultSet.
- Both program styles are designed to minimize the code clutter, code segments are contained either within a try{ } block or within the DownRow() method.
- The try{ } block is where the SQL statement is created and executed.
- The **DownRow()** method is where the J2ME application interacts with the ResultSet.
- the executeUpdate() method is used to execute the query, and no ResultSet is returned. Therefore, you can replace the second try { } block in the Model A program with the code segment, the Model Aprogram does not contain a DownRow() method because no ResultSet is returned.
- If we want to retrieve data from the database.The executeQuery() method is used in this scenario.
- The ModelB program is used for this purpose because a ResultSet is returned. to retrieve data from a database contains two code segments.
- The first code segment contains the query, and it replaces the second try{ }block in the ModelB program.
- The other code segment replaces the DownRow() method of the Model B program and interacts with the ResultSet.

##### Model A Program:

- The Model Aprogram, is designed to execute queries that do not return a ResultSet.
- It is organized into the ModelA() constructor and the main().
- The main() creates an instance of the ModelAobject called sql1, which causes the execution of the constructor.

- **The constructor begins by creating three String objects.**
- **These are url, userId, and password. The url is assigned the URL of the database.**
- The first try {} block in ModelA uses the **Class.forName()** method to load the JDBC driver into the JavaVirtualMachine(JVM).
- Once the JDBC driver is loaded, the **getConnection()** method is called to open a connection to the database using the url, userID, and password String objects.
- The **getConnection()** method returns a Connection object called database, which is declared above the definition of the constructor.
- **Two catch {} blocks follow**
- **the first try {} block** and are used to trap exceptions that occur while the driver is loaded and a connection is established with the database.
- **The second try{} block** in ModelA contains a comment.

### Example:

```
import java.sql.*;

public class ModelA
{
    private Connection Database;
    private Statement DataRequest;

    public ModelA ()
    {
        String url = "jdbc:odbc:CustomerInformation";
        String userID = "jim";
        String password = "keogh";

        try
        {
            Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
            Database = DriverManager.getConnection(url,userID,password);
        }
        catch (ClassNotFoundException error)
        {
            System.err.println("Unable to load the JDBC/ODBC bridge." + error);
        }
    }
}
```

```
System.exit(1);
}
catch (SQLException error)
{
System.err.println("Cannot connect to the database. "+ error);
if(Database != null)
{
try
{
Database.close();
}
catch(SQLException er){ }
}
System.exit(2);
}
Try
{
// insert example code here
}
catch ( SQLException error )
{
System.err.println("SQL error." + error);
if(Database != null)
{
Try
{
Database.close();
}
}
```

```
catch(SQLException er){ }  
}  
System.exit(3);  
}  
if(Database != null)  
{  
try  
{  
Database.close();  
}  
catch(SQLException er){ }  
}  
}  
public static void main ( String args [] )  
{  
final ModelA sql1 = new ModelA ();  
System.exit ( 0 ) ;  
}  
}
```

### **Model B Program:**

- The Model B program is designed for use by J2ME components that retrieve information from a database.
- the Model B program is similar to the Model A program in that the constructor definition (except for the name of the constructor) and the main() are identical.
- The try { } executes a query and returns a ResultSet object called Results, which is declared as a private member of the ModelB class.
- Model B also contains DisplayResults() and DownRow(). DisplayResults() is passed the ResultSet returned in the second try{ } block, which is used to move the virtual cursor to the first row of the ResultSet using the next() method.
- The next() returns a boolean value that is assigned to the Records variable.

- The if statement evaluates the value of Records and displays a “No data returned” message if Records contains a false value indicating that there isn’t any data in the row.
- A true value causes the do...while loop to execute, which is where a call to DownRow() is made, passing it the ResultSet.
- DownRow() retrieves data stored in columns of the ResultSet and assigns those values to variables.
- AfterDownRow()extractsdatafromthecurrentrowoftheResultSet,controlreturns toDisplayResults(),wherethenext()methodiscalledandtheresultsareevaluatedbythe while.
- If the next() method returns a true value, then DownRow() is recalled, otherwise the do...while loop is exited and control returns to the constructor.
- DisplayResults()alsocontainsacatch{ } blockthattraperrorsthrownbystatements within the try { } block.

```
import java.sql.*;

public class ModelB
{
    private Connection Database;
    private Statement DataRequest;
    private ResultSet Results;

    public ModelB () { String url = "jdbc:odbc:CustomerInformation"; String userID = "jim";
String password = "keogh"; try { Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver"); Database =
DriverManager.getConnection(url,userID,password); } catch (ClassNotFoundException error) {
System.err.println("Unable to load the JDBC/ODBC bridge." + error); System.exit(1); } catch
(SQLException error) { System.err.println("Cannot connect to the database." + error);
System.exit(2); } try { // Enter example code here } catch ( SQLException error ) {
System.err.println("SQL error." + error); if(Database != null) { try { Database.close(); }
catch(SQLException er){ } } System.exit(3); } if(Database != null) {

try { Database.close(); } catch(SQLException er){ } }

} private void DisplayResults (ResultSet DisplayResults) throws SQLException { boolean
Records = DisplayResults.next(); if (!Records ) { System.out.println( "No data returned"); return;
} try { do { DownRow( DisplayResults) ; } while ( DisplayResults.next() ); } catch
(SQLException error ) { System.err.println("Data display error." + error); if(Database != null) {
try { Database.close(); } catch(SQLException er){ } } System.exit(4); } } private void DownRow
( ResultSet DisplayResults ) throws SQLException { //Enter new DownRow() code here } public
static void main ( String args [] ) { final ModelB sql1 = new ModelB (); System.exit(0); }

}
```

### **TABLES:**

- A database design is used as the basis for building tables and indexes that make up a database.
- Tables and indexes are created using a query written using SQL. Typically, the database administrator writes and executes the query that creates tables and indexes, although it is possible to have a J2ME application or a Java application execute the same query to create tables and indexes.

### Create a Table:

- The query contains the **CREATE TABLE SQL** statement that contains the name of the table.
- The datatype of the column and the column size follow the column name. A comma separates each column.
- All the SQL commands are enclosed in double quotation marks and assigned to a String object called query.
- Next, the program creates an SQL statement using the **createStatement()** method of the Connection object.
- This method returns the handle to the DBMS to the **DataRequest object**, which is an object of the Statement class.
- The query is sent to the DBMS using the **executeQuery()** method.
- If successful, the DBMS creates the table.
- If unsuccessful, the table isn't created and the DBMS returns an error message.
- The SQL statement is then closed using **DataRequest.close()**.

```
try { String query = "CREATE TABLE CustomerAddress ( " + " CustomerNumber CHAR(30), " + " CustomerStreet CHAR(30), " + " CustomerCity CHAR(30), " + " CustomerZip CHAR(30))"; DataRequest = Database.createStatement(); DataRequest.executeQuery(query); DataRequest.close(); }
```

### Requiring Data in a Column:

- There are business rules that require a value in specific columns of a row, such as the columns that contain a customer's name.
  - can require the DBMS to reject rows that are missing a value in specific columns by using the **NOT NULL clause** in the query when the table is created.
- ```
try { String query = "CREATE TABLE CustomerAddress ( " + "CustomerNumber CHAR(30) NOT NULL," + "CustomerStreet CHAR(30) NOT NULL," + "CustomerCity CHAR(30) NOT NULL," + "CustomerZip CHAR(30) NOT NULL)"; DataRequest = Database.createStatement(); DataRequest.executeQuery(query); DataRequest.close(); }
```

### Setting a Default Value for a Column :

- The DBMS can enter a default value into a column automatically if the column is left empty whenever a new row is inserted into the table.



- can determine the value entered by the DBMS by creating a default value when you create the table.
- Any value can be used as the default value as long as the value conforms to the data type and size of the column.
- A default value is assigned to a column when you create the table by using the **DEFAULT clause**.
- The **DEFAULT clause** is followed in the query by the value that the DBMS will place in the column if the column is empty in the incoming row.

```
try { String query = "CREATE TABLE CustomerAddress ( " + "CustomerNumber CHAR(30) NOT NULL," + "CustomerStreet CHAR(30) NOT NULL," + "CustomerCity CHAR(30) NOT NULL," + "CustomerZip CHAR(30) NOT NULL DEFAULT '07660'"; DataRequest = Database.createStatement(); DataRequest.executeUpdate(query); DataRequest.close(); }
```

### Drop a Table:

- A developer may have the right to remove a table, but this is usually reserved for the development environment only.
- The decision to drop a table shouldn't be made lightly because once a table is dropped you cannot recover it.
- Instead, the table must be re-created and the data re inserted into the table.
- In addition to losing data elements stored in the table, dropping a table may affect the integrity of the database and tables that relate to values in the dropped table.
- Using the DropTable statement in the query drops a table.  

```
try { String query = new String ("DROP TABLE CustomerAddress"); DataRequest = Database.createStatement(); DataRequest.executeUpdate(query); DataRequest.close(); }
```

### INDEXING:

- The database schema describes indexes along with tables that are used in the database.
- Executing the query to create or drop an index follows procedures similar to those for creating and dropping a table.
- That is, the query can be executed within or outside a J2ME application or Java application.

### Create an Index:

- An index is created by using the **CREATE INDEX** statement in a query.
- The **CREATE INDEX** statement contains the name of the index and any modifier that describes to the DBMS the type of index to be created.
- In addition, the **CREATE INDEX** statement uses the **ON** clauses to identify the name of the table and the name of the column whose value is used for the index.

- creates an index called CustNum that contains values in the CustomerNumber column of the CustomerAddress table.
- The CustNum index doesn't have duplicate key values, and therefore the **UNIQUE** modifier is used in the **CREATE INDEX** statement to prohibit duplicate key values from being included in the index.

```
try { String query = "CREATE UNIQUE INDEX CustNum " + "ON CustomerAddress  
(CustomerNumber)"; DataRequest = Database.createStatement(); DataRequest.execute (query);  
DataRequest.close(); }
```

### Designating a Primary Key:

- A primary key can be designated when a table is created by using the Primary Key modifier.
- This example contains a query that creates a table and one column within the table is designated the primary key. That is, the column cannot contain duplicate values.
- the **OrderNumber** column is used as the primary key and is identified in the **PRIMARYKEY** modifier.
- Once a column is designated as the primary key, the DBMS prevents duplicate and null values from being placed in the column.

```
try { String query = "Create Table Orders ( " + "OrderNumber CHAR(30) NOT NULL, " +  
"CustomerNumber CHAR(30), " + "ProductNumber CHAR(30), " + "CONSTRAINT  
ORDERS_PK PRIMARY KEY (OrderNumber)"; DataRequest = Database.createStatement();  
DataRequest.execute(query); DataRequest.close(); }
```

### Creating a Secondary Index:

- A secondary index is created by using the **CREATEINDEX** statement in a query without the use of the **UNIQUE** modifier.
- This means that a secondary index can have duplicate values.  

```
try { String query = new String ("CREATE INDEX CustZip " + "ON CustomerAddress  
(CustomerZip) "); DataRequest = Database.createStatement(); DataRequest.  
execute(query); DataRequest.close(); }
```

### Creating a Clustered Index:

- A clustered index is an index whose key is created from two or more columns of a table.
- For example, combining a customer's last name with a customer's first name is a typical key for a clustered index.
- This example creates an index called CustName and uses the LastName and FirstName columns of the Customers table as the key for the index.
- The order in which column names appear in the **ON** clause plays a critical role in the index

- The value of the second column is concatenated to the value of the first column that appears in the query.  
try { String query = "CREATE INDEX CustName " + " ON Customers (LastName, FirstName)"; DataRequest = Database.createStatement(); DataRequest.execute (query); DataRequest.close(); }

### Drop an Index :

- An existing index can be removed from the database by using the **DROP INDEX** statement  
try { String query = new String("DROP INDEX CustName ON Customers "); DataRequest = Database.createStatement(); DataRequest. execute(query); DataRequest.close(); }

### INSERTING DATA INTO TABLES

#### Insert a Row :

- The **INSERT INTO** statement is used to insert a new row into a table.
- The **INSERT INTO** statement contains the name of the table into which the row is to be inserted and the name of the columns in which values are inserted.
- The **VALUES clause** is used to define the values to be placed into the row.

```
try { String query = "INSERT INTO Customers " + " (CustomerNumber, FirstName, LastName, DateOfFirstOrder) " + " VALUES (1,'Mary','Smith','10/10/2001') "; DataRequest = Database.createStatement(); DataRequest.executeUpdate (query); DataRequest.close(); }
```

#### Insert the System Date into a Column:

- place the system date into a column by calling the **CURRENT\_DATE** function.
- Oracle uses SYSDATE, and DB2 uses CURRENT DATE (no underscore).
- The **CURRENT\_DATE** function directs the DBMS to use the system date of the server as the value for the column.

```
try { String query = new String ( "INSERT INTO Customers (CustomerNumber ,FirstName, "+ " LastName, DateOfFirstOrder )" + " VALUES ( 4,'Mary','Jones', CURRENT_DATE)"); DataRequest = Database.createStatement(); DataRequest.executeUpdate (query) ; DataRequest.close(); }
```

#### Insert the System Time into a Column:

- Call the **CURRENT\_TIME()** function whenever a column requires the current time.
- The **CURRENT\_TIME()** function is called by the DBMS when the query is processed, and it returns the current time of the server.

```
try { String query = new String ( "INSERT INTO Customers (CustomerNumber ,FirstName, "+ " LastName, TimeOfFirstOrder ) " + " VALUES ( 2,'Bob','Jones', CURRENT_TIME() ) ) ;
```

```
DataRequest = Database.createStatement(); DataRequest.executeUpdate(query );
DataRequest.close(); }
```

### Insert a Timestamp into a Column:

- A timestamp consists of both the current date and time and is used in applications where both date and time are critical to the operation of the business.

```
try { String query = new String ( "INSERT INTO Customers (CustomerNumber ,FirstName, "+ "
LastName, FirstOrder ) " + " VALUES ( 2,'Bob','Jones', CURRENT_TIMESTAMP() )");
DataRequest = Database.createStatement(); DataRequest.executeUpdate (query );
DataRequest.close(); }
```

### SELECTING DATA FROM A TABLE

- Retrieving information from a database is the most frequently used routine of J2ME components that interact with a database.

#### Select All Data from a Table:

- The **SELECT** statement is used to retrieve data from a table.
- The **executeQuery()** method, which is used to execute the query, returns a **ResultSet** object.
- The **ResultSet** object is passed to the **DisplayResults()** method.
- The **DisplayResults()** method is responsible for moving through the **ResultSet** and displaying the contents of the **ResultSet** on the screen.
- The **DownRow()** method then declares **String** objects that are assigned the value of each column in the **ResultSet**.
- There is also a **String** object called **printrow**, which is assigned all the other **String** objects and is used to display the values on the screen.
- The **getString()** method is called to gather the value of a specific column in the **ResultSet**, The **getString()** method returns the value of the column, which is assigned to a **String** object.

```
try { String query = new String ("SELECT " + " FirstName, LastName, Street, City, State,
ZipCode " + " FROM Customers"); DataRequest = Database.createStatement(); Results =
DataRequest.executeQuery (query); DisplayResults (Results); DataRequest.close(); }
```

```
private void DownRow ( ResultSet DisplayResults ) throws SQLException { String FirstName=
new String(); String LastName= new String(); String Street= new String(); String City = new
String(); String State = new String(); String ZipCode= new String(); String printrow; FirstName
= DisplayResults.getString ( 1 ) ; LastName = DisplayResults.getString ( 2 ) ; Street =
DisplayResults.getString ( 3 ) ; City = DisplayResults.getString ( 4 ) ; State =
DisplayResults.getString ( 5 ) ; ZipCode = DisplayResults.getString ( 6 ) ; printrow = FirstName
+ " " + LastName + " " + City + " " + State + " " + ZipCode; System.out.println(printrow); }
```

### Request One Column:

- can specify a column that you want returned from the table by using the column name in the **SELECT** statement
- the Last Name column of all the rows in the Customers table is returned in the ResultSet.
- the **DownRow()** method that is used to retrieve data from the ResultSet returned by

```
try { String query = new String ("SELECT LastName FROM Customers"); DataRequest =  
Database.createStatement(); Results = DataRequest.executeQuery (query);
```

```
DisplayResults (Results); DataRequest.close();
```

```
}
```

```
private void DownRow ( ResultSet DisplayResults )throws SQLException { String LastName=  
new String(); LastName = DisplayResults.getString(1); System.out.println(LastName); }
```

### Request Multiple Columns

- Multiple columns can be retrieved by specifying the names of the columns in the **SELECT** statement similar to the way you select one column.
- where the FirstName and LastName columns for all the rows in the Customers table are returned in the ResultSet.
- The **DownRow()** method that is used to copy and display values from the ResultSet.

```
try { String query = new String ( "SELECT FirstName, LastName FROM Customers");  
DataRequest = Database.createStatement(); Results = DataRequest.executeQuery (query);  
DisplayResults (Results); DataRequest.close(); }
```

```
private void DownRow ( ResultSet DisplayResults )throws SQLException { String FirstName=  
new String(); String LastName= new String(); String printrow; FirstName =  
DisplayResults.getString ( 1 ) ; LastName = DisplayResults.getString ( 2 ) ; printrow =  
FirstName + " " + LastName; System.out.println(printrow); }
```

### Request Rows:

- Specific rows can be retrieved from a column by using the **WHERE** clause in conjunction with the **SELECT** statement.
- The **WHERE** clause contains an expression that is used by the DBMS to identify rows that should be returned in the ResultSet.
- Any logical expression can be used to include or exclude rows based on values in columns of a row.
- In this example, the **SELECT** statement retrieves all the columns of rows in the Customers table where the LastName column has the value 'Jones'.

```
try { String query = new String ("SELECT " + " FirstName, LastName, Street, City,  
State, ZipCode " + " FROM Customers " + " WHERE LastName = 'Jones' ");  
DataRequest = Database.createStatement(); Results = DataRequest.executeQuery  
(query); DisplayResults (Results); DataRequest.close(); }
```

### Request Rows and Columns:

- A query can select fewer than all the columns and all the rows of a table by using a combination of techniques
- In this example, the FirstName and LastName of rows in the Customers table are returned in the ResultSet as long as the value of the LastName column is 'Jones'.

```
try { String query = new String("SELECT FirstName, LastName " + " FROM Customers  
" + " WHERE LastName = 'Jones' "); DataRequest = Database.createStatement();  
Results = DataRequest.executeQuery (query); DisplayResults (Results);  
DataRequest.close(); }
```

### AND, OR, and NOT Clauses:

- The **WHERE** clause in a **SELECT** statement can evaluate values in more than one column of a row by using the **AND**, **OR**, and **NOT** clauses to combine expressions.
- compound expressions in the **WHERE** clause are individually evaluated, and then the results of these evaluations are further evaluated based on the **AND**, **OR**, and **NOT** clauses in the **WHERE** clause.
- the **AND** clause requires that both expressions in the compound expression evaluate to true before the **WHERE** clause expression evaluates true and includes a specified row in the ResultSet.
- The **OR** clause requires that at least one of the expressions in the compound expression evaluate to true before the **WHERE** clause expression evaluates true.
- And the **NOT** clause is used to reverse the logic, changing an expression that evaluates true to a false.

### AND Clause

- The purpose of the AND clause is to join two subexpressions together to form one compound expression.
- The AND clause tells the DBMS that the boolean value of both sub expressions must be true for the compound expression to be true.
- If the compound expression is true, the current row being evaluated by the DBMS is returned to your program.
- the FirstName and LastName columns of rows from the Customers table are returned in the ResultSet only if the value of the LastName column is 'Jones' and the value of the FirstName column is 'Bob'.

```
try { String query = new String ("SELECT FirstName, LastName " + " FROM Customers  
" + " WHERE LastName = 'Jones' " + " AND FirstName = 'Bob'"); DataRequest =  
Database.createStatement(); Results = DataRequest.executeQuery (query);  
DisplayResults (Results); DataRequest.close(); }
```

### OR Clause

- The **OR** clause is used to create a compound expression using two sub expressions in the sameway as the **AND** clause.
- The OR clause tells the DBMS that the compound expression evaluates to a boolean true if either of the two subexpressions evaluates to a boolean true.
- The FirstName and LastName columns of rows from the Customers table are returned in the ResultSet only if the value of the First Name column is 'Mary' or 'Bob'.

```
try { String query = new String ("SELECT FirstName, LastName " + " FROM Customers  
" + " WHERE FirstName = 'Mary' " + " OR FirstName = 'Bob'"); DataRequest =  
Database.createStatement(); Results = DataRequest.executeQuery (query);  
DisplayResults (Results); DataRequest.close(); }
```

### NOT Clause:

- The **NOT** clause reverses the logic of the subexpression contained in the **WHERE NOT** clause.
- If the subexpression evaluates to a boolean true, the **NOT** clause reverses the logic to return a boolean false.
- In contrast, if the subexpression evaluates to a boolean false, the compound expression evaluates to a boolean true

```
try { String query = new String( "SELECT FirstName, LastName " + "FROM Customers  
" + "WHERE NOT FirstName = 'Mary' " ); DataRequest = Database.createStatement();  
Results = DataRequest.executeQuery (query); DisplayResults (Results);  
DataRequest.close(); }
```

### Join Multiple Compound Expressions:

- The **AND** and **OR** clauses are used to link together two or more subexpressions, which results in a compound expression.
- There can be multiple compound expressions within a **WHERE** clause expression.
- can format a query that uses both the **AND** clause and the **OR** clause to retrieve this information.

**WHERE** FirstName = 'Bob' **AND** LastName = 'Smith **AND** ( Dept = '42' OR Dept = '45')

- Although the **WHERE** clause may appear confusing, can simplify the expression by identifying subexpressions and compound expressions

FirstName = 'Bob' **AND** LastName = 'Smith'

- The second compound expression is



Dept = '42' OR Dept = '45'

- The second **AND** clause in the **WHERE** clause links these two compound expressions.
- Can use the **AND**, **OR**, and **NOT** clauses to create complex selection expressions.
- The number of compound expressions that you can use in the **WHERE** clause is practically endless, although some DBMSs establish a limit.
- This is because of processing time that is necessary to evaluate a complex **WHERE** clause expression.

```
try { String query = new String ("SELECT " + "FirstName, LastName, Street, City, State, ZipCode, Sales " + "FROM Customers " + "WHERE NOT Sales = 50000 " ); DataRequest = Database.createStatement(); Results = DataRequest.executeQuery (query); DisplayResults (Results); DataRequest.close(); Database.close(); }
```

```
private void DownRow ( ResultSet DisplayResults )throws SQLException { String FirstName=new String(); String LastName= new String(); String Street= new String(); String City = new String(); String State = new String(); String ZipCode= new String(); long Sales; String printrow; FirstName = DisplayResults.getString ( 1 ) ;
```

```
LastName = DisplayResults.getString ( 2 ) ; Street = DisplayResults.getString ( 3 ) ; City = DisplayResults.getString ( 4 ) ; State = DisplayResults.getString ( 5 ) ; ZipCode = DisplayResults.getString ( 6 ) ; Sales = DisplayResults.getLong ( 7 ) ; printrow = FirstName + " " + LastName + " " + City + " " + State + " " + ZipCode + " " + Sales; System.out.println(printrow); }
```

### Less Than and Greater Than Operators:

- The less than and greater than operators direct the DBMS to assess whether or not the value in the specified column of the current row is less than or greater than the value in the **WHERE** clause expression.
- A value in a column that equals the value in the **WHERE** clause expression is evaluated as a Boolean false. Therefore the row containing that value isn't returned in the ResultSet.
- The value in the column must be less than or greater than but not equal to the value in the **WHERE** clause expression.

### Less Than Operator:

- all the columns of rows from the Customers table are returned as long as the value of the Sales column is less than 50000.
- the DownRow() method that should be used to copy and display the ResultSet returned by  

```
try { String query = new String ("SELECT " + "FirstName, LastName, Street, City, State, ZipCode, Sales " + " FROM Customers " + " WHERE Sales < 50000 " );
```



```
DataRequest = Database.createStatement(); Results = DataRequest.executeQuery  
(query); DisplayResults (Results); DataRequest.close(); }
```

### Greater Than Operator :

- The greater than operator is similar to the less than operator, except the value in the column must be greater than the value in the **WHERE** clause expression for the row to be returned in the **ResultSet**.
- all the columns of rows in the Customers table are returned in the **ResultSet** if the value of the Sales column is greater than 40000.

```
try { String query = new String ("SELECT " + "FirstName, LastName, Street, City, State,  
ZipCode, Sales " + "FROM Customers " + "WHERE Sales > 40000 " ); DataRequest =  
Database.createStatement(); Results = DataRequest.executeQuery (query); DisplayResults  
(Results); DataRequest.close(); }
```

### Less Than or Equal to and Greater Than or Equal To:

- A drawback of using the less than and greater than operators is that rows containing the value of the **WHERE** clause expression are not returned in the **ResultSet**.
- Alternatively, the less than or equal to and the greater than or equal to operators can be used to include rows that contain the **WHERE** clause expression.

### Less Than or Equal To :

- all columns of rows in the Customers table are returned if the value of the Sales column of the row is less than or equal to 50000.
- the **DownRow()** method that should be used to copy and display the **ResultSet** returned

### Greater Than or Equal To:

- the greater than or equal to operator is similar to the less than or equal to operator.
- All the columns in rows of the Customers table are returned in the **ResultSet** if the value of the Sales column in the row is greater than or equal to 50000.
- the **DownRow()** method that should be used to copy and display the **ResultSet** returned .

```
try { String query = new String ("SELECT " + "FirstName, LastName, Street, City, State,  
ZipCode, Sales " + "FROM Customers " + "WHERE Sales >= 50000 "); DataRequest =  
Database.createStatement(); Results = DataRequest.executeQuery (query); DisplayResults  
(Results); DataRequest.close(); }
```

### Between Operator:

- The **BETWEEN** operator is used to define a range of values to be used as the value of the selection expression.
- The range must consist of a sequential series of values, such as 100 to 200.

- The **BETWEEN** operator must follow the name of the column in the **WHERE** clause.
- The **AND** operator is used to join the lower and upper values of the range. All values in the range, including the first and last values, are considered when the DBMS evaluates the value of the column specified in the selection expression.
- all the columns in rows of the Customers table are returned in the ResultSet if the value of the Sales column in the row is between 20000 and 39999.

```
try { String query = new String("SELECT " + "FirstName, LastName, Street, City, State, ZipCode, Sales " + "FROM Customers " + "WHERE Sales BETWEEN 20000 AND 39999 " ); DataRequest = Database.createStatement(); Results = DataRequest.executeQuery (query); DisplayResults (Results); DataRequest.close(); }
```

```
try { String query = new String ("SELECT " + "FirstName, LastName, Street, City, State, ZipCode, Sales " + "FROM Customers " + "WHERE Sales <= 50000 " ); DataRequest = Database.createStatement(); Results = DataRequest.executeQuery (query); DisplayResults (Results); DataRequest.close(); }
```

### LIKE Operator:

- The **LIKE** operator directs the DBMS to return a row in the ResultSet if a value in a specified column partially matches the value of the **WHERE** clause expression.
- The **WHERE** clause expression must include a character that is an exact match and a wildcard character that is used to match any other character.
  - **Underscore ( \_ )** A single-character wildcard character. For example, if you are unsure whether the customer's last name is Anderson or Andersen, use the underscore in place of the character that is in question, as in Anders\_n.
  - **Percent ( % )** A multicharacter wildcard character used to match any number of characters. For example, Smi% is used to match a value of a column where the first three characters are Smi followed by any other character(s).

```
try { String query = new String ("SELECT " + "FirstName, LastName, Street, City, State, ZipCode, Sales " + "FROM Customers " + "WHERE LastName LIKE 'Smi%' "); DataRequest = Database.createStatement(); Results = DataRequest.executeQuery (query); DisplayResults (Results); DataRequest.close(); }
```

### IS NULL Operator :

- The **IS NULL** operator is used to determine whether a specified column does not contain any value.
- a column that contains a space isn't NULL because a space is a valid ASCII character.
- **NULL** is void of any value and occurs when a row is inserted into a table without having a value, or the value is explicitly set to NULL.

```
try { String query = new String ("SELECT " + "FirstName, LastName, Street, City, State, ZipCode, Sales " + "FROM Customers " + "WHERE State IS NULL ");
```

```
DataRequest = Database.createStatement(); Results = DataRequest.executeQuery  
(query); DisplayResults (Results); DataRequest.close(); }
```

### **DISTINCT Modifier :**

- The **SELECT** statement returns all rows in a table unless a **WHERE** clause is used to excludespecificrows.
- theResultSetincludesduplicaterowsunlessaprimariy  
indexiscreatedforthetableoronlyuniquerowsarerequiredinthetable.
- Therewillbeoccasionswhenyouwanttoexcludeallbutonecopyofarowfromthe ResultSet. can  
do this by using the **DISTINCT** modifier in the **SELECT** statement.
- The **DISTINCT** modifier tells the DBMS not to include duplicate rows in the ResultSet.  
will find this useful whenever data from multiple sources is combined into one table, as  
in a mailing list.
- The **DISTINCT** modifier filters duplicate rows that contain the same name and address.
- An alternative approach is to write a **WHERE** clause expression that returns the same  
results as using the **DISTINCT** modifier  
try { String query = new String ("SELECT DISTINCT " + "FirstName, LastName,  
Street, City, State, ZipCode, Sales " + "FROM Customers "); DataRequest =  
Database.createStatement(); Results = DataRequest.executeQuery (query);  
DisplayResults (Results); DataRequest.close(); }

### **IN Modifier**

- The **IN** modifier is used to define a set of values used by the DBMS to match values in a  
specified column.
- The set can include any number of values and appear in any order.
- place the three product identification numbers ina set;then tell theDBMS to search the  
order table for orders containing the product numbers in the set.
- If the product numbers match, customer information associated with the row is returned  
to your program.
- The **IN** modifier is used in the **WHERE** clause to define the list of values in the set.  
try { String query = new String ("SELECT " + "FirstName, LastName, Street, City,  
State, ZipCode, Sales " + " FROM Customers " + " WHERE Sales IN (20000, 30000,  
40000) " ); DataRequest = Database.createStatement(); Results =  
DataRequest.executeQuery (query); DisplayResults (Results); DataRequest.close(); }

### **NOT IN Modifier**

- The **NOT IN** modifier is similar to the **IN** modifier,except the **NOT IN** modifier reverses  
the logic. That is, it identifies a set of values that shouldn't match rows returned to the  
program.

```
try { String query = new String ("SELECT " + "FirstName, LastName, Street, City, State,  
ZipCode, Sales " + " FROM Customers " + " WHERE Sales NOT IN (20000, 30000, 40000) " );  
DataRequest = Database.createStatement(); Results = DataRequest.executeQuery (query);  
  
DisplayResults (Results); DataRequest.close();  
  
}
```

### Metadata

- Metadata is data that describes data. Metadata is returned with the **ResultSet** object and can be extracted from the **ResultSet** object by creating a **ResultSetMetaData**.
- Metadata can be used in a J2ME application for various purposes, such as to display the column name of a column and determine the data type of a column.
- The most commonly used metadata are
  - Column name
  - Column number
  - Column data type
  - Column width

### Number of Columns in ResultSet:

- The **getMetaData()** method is called to extract metadata from the **ResultSet**.
- The **getMetaData()** method returns a **ResultSetMetaData** object, which is called metadata
- The **ResultSetMetaData** object contains several methods that are used to copy specific metadata from the **ResultSet**.
- The **getColumnCount()** method is called to retrieve the number of columns contained in the result set, which is then assigned to the **NumberOfColumns** variable and printed on the screen using the **println()** method.
- The **DownRow()** method in the Model B program and be used in any version of the Model B program. Of course, the **DownRow()** method must be modified to copy and display data contained in the **ResultSet**  

```
private void DownRow ( ResultSet DisplayResults )throws SQLException {  
    ResultSetMetaData metadata = DisplayResults.getMetaData (); int NumberOfColumns;  
    String printrow; NumberOfColumns = metadata.getColumnCount ();  
    System.out.println("Number Of Columns: " + NumberOfColumns); }
```

### Data Type of Column

- This technique is very similar to the technique for retrieving the number of columns in the ResultSet .
  - The **getColumnTypeName()** method is called to copy the data type from column nine of the result set to the ColumnType object.
  - Any valid column number in the ResultSet can be passed to the **getColumnTypeName()** method
- ```
private void DownRow ( ResultSet DisplayResults ) throws SQLException {  
    ResultSetMetaData metadata = DisplayResults.getMetaData (); String ColumnType =  
    new String(); String printrow; ColumnType = metadata.getColumnTypeName ( 9 );  
    System.out.println("Column Type: " + ColumnType ); }
```

### Name of Column:

- Retrieving the column name from the metadata uses a process similar to copying the data type of a column from the metadata.
- The **getColumnLabel()** method is used to copy the column name from a specific column and assign the column name to the String object ColumnName.
- The number of the column in the ResultSet is passed to the **getColumnLabel()** method.

```
private void DownRow ( ResultSet DisplayResults ) throws SQLException {  
    ResultSetMetaData metadata = DisplayResults.getMetaData (); String ColumnName = new  
    String(); String printrow; ColumnName = metadata.getColumnLabel (9) ;  
    System.out.println("Column Name: " + ColumnName); }
```

### Column Size

- The column size, also referred to as the column width, is called the display size and represents the number of characters needed to display the maximum value that might be stored in the column.
- Can retrieve the display size by using the **getColumnDisplaySize( )** method.
- The **getColumnDisplaySize( )** is called and passed the number of the column whose display size is to be retrieved from the ResultSet.

```
private void DownRow ( ResultSet DisplayResults ) throws SQLException {  
    ResultSetMetaData metadata = DisplayResults.getMetaData (); int ColumnWidth; String  
    printrow; ColumnWidth = metadata.getColumnDisplaySize ( 9 ) ; System.out.println("Column  
    Width:" + ColumnWidth); }
```

### Updating Tables :

- Modifying data in a database is one of the most common functionalities in every J2ME application that provides database interactions.

- Generally, any information that is retrievable is also changeable depending on access rights and data integrity issues.
- The **executeUpdate()** method to process queries.
- The **executeUpdate()** method does not return a **ResultSet**. Therefore, the **Model** Aprogram should be used with code segments.
- The **UPDATE** statement is used to change the value of one or more columns in one or multiple rows of a table.
- The **UPDATE** statement must contain the name of the table thatistobeupdatedanda**SET**clause.The **SET**clause identifies the name of the column and the new value that will be placed into the column, overriding the current value.
- The **UPDATE** statement may have a **WHERE** clause if a specific number of rows are tobe updated.
- If the **WHERE** clause is omitted,all rows are updated based on the value of the **SET** clause.

```
try { String query = new String("UPDATE Customers " + "SET Street = '5 Main Street' " + " WHERE FirstName = 'Bob'"); DataRequest = Database.createStatement(); DataRequest.executeUpdate (query); DataRequest.close(); }
```

### Update Multiple Rows

- Multiple rows of a table can be updated by formatting the **WHERE** clause expressions to include criteria that qualify multiple rows for the update.
- Four common **WHERE** clause expressions are used to update multiple rows of a table:
  - The **IN** test The **WHERE** clause expression contains multiple values in the **IN** clausethat must match the value in the specified column for the update to occur in the row.
  - The **IS NULL** test Rows that don't have a value in the specified column are updated when the **IS NULL** operator is used in the **WHERE** clause expression.
  - The comparison test The **WHERE** clause expression contains a comparison operator, that compares the value in the specified column with a value in the **WHERE** clause expression.
  - All rows Aquery can direct the DBMS to update the specified column in all rows of a table by excluding the **WHERE** clause in the query.
- An error in a query is multiplied by the number of rows in a table when ever the query updates more than one row.
- **IN** Test The **IN** clause provides two or more values that are compared to the value of the designated column in the **IN** clause.
- Rows whose columns contain one of these values are updated by the **UPDATE** statement.

### Update Based on Values in Column:

- An expression in the **WHERE** clause can be used to identify rows that are to be updated by the **UPDATE** statement.
- All of the **WHERE** clause expressions discussed previously in the **SELECT** statement also apply to the **UPDATE** statement.
- Review the section “Selecting Data from a Table” for details on how to properly formulate the **WHERE** clause expression.

```
try { String query = new String ("UPDATE Customers " + "SET Discount = 20 " +  
"WHERE Discount > 20 "); DataRequest = Database.createStatement();  
DataRequest.executeUpdate (query); DataRequest.close(); }
```

### Update Every Row

- All rows in a table can be updated by excluding the **WHERE** clause in the **UPDATE** statement.
- the value of the Discount column in all rows in the Customers table is changed to zero.

```
try { String query = new String ("UPDATE Customers " + "SET Discount = 0 "); DataRequest =  
Database.createStatement(); DataRequest.executeUpdate (query); DataRequest.close(); }
```

### Update Multiple Columns

- Multiple columns of rows can be updated simultaneously by specifying the column names and appropriate values in the **SET** clause of the query.
- The **SET** clause contains the column names and the new values that override the current values of those columns. In this case, the value of the Discount column is changed to 12, and the value of the Street column is changed to ‘Jones Street’.

```
try { String query = new String ("UPDATE Customers " + "SET Discount = 25 " + "WHERE  
Discount IN (12,15)"); DataRequest = Database.createStatement(); DataRequest.executeUpdate  
(query); DataRequest.close(); }
```

### IS NULL Test

- The **IS NULL** test evaluates the value of a column designated in the test to determine whether the column is NULL—that is, the column is empty of any value.
- If so, the **IS NULL** test returns a true, and the **UPDATE** statement updates the column specified in the **SET** clause.

```
try { String query = new String ("UPDATE Customers " + "SET Discount = 0 " +  
"WHERE LastName IS NULL "); DataRequest = Database.createStatement();  
DataRequest.executeUpdate (query); DataRequest.close(); }
```

```
try { String query = new String ("UPDATE Customers " + "SET Discount = 12, Street = 'Jones  
Street'" + "WHERE LastName = 'Jones'"); DataRequest = Database.createStatement();  
DataRequest.executeUpdate (query); DataRequest.close(); }
```



### Update Using Calculations

- The value that replaces the current value in a column does not have to be explicitly defined in the **SET** clause if the value can be derived from a value in an other column of the same row.
- The customer is granted a percentage discount based on how well the customer is valued by the business.
- The **UPDATE** statement can calculate the discounted price and place it in the DiscountPrice column of the row.
- The value of the discount price doesn't need to be included in the SET clause. Instead, the discount price can be calculated.

```
try { String query = new String ("UPDATE Customers " + "SET DiscountPrice = Price *  
((100 - Discount) / 100) "); DataRequest = Database.createStatement();  
DataRequest.executeUpdate (query); DataRequest.close(); }
```

### Deleting Data from a Table

- Deleting rows is necessary to purge erroneous information from the database and to remove information that is no longer needed.
- Before we delete a row from a table, you must be certain that other tables are not negatively affected.
- for removing a row from a table by using the **DELETE FROM** statement.
- Multiple rows can be deleted by including a **WHERE** clause in the **DELETE FROM** statement.
- The query that contains the **DELETE FROM** statement is executed using the **executeQuery()** method.
- The **executeQuery()** method doesn't return a ResultSet.
- Therefore, use the ModelA program with the code segment.
- Delete a Row from a Table The **DELETE FROM** statement includes the name of the table and a **WHERE** clause containing an expression that identifies the row or rows to remove from the table.

```
try { String query = new String ("DELETE FROM Customers " + "WHERE LastName =  
'Jones' and FirstName = 'Tom'"); DataRequest = Database.createStatement();  
DataRequest.executeUpdate (query); DataRequest.close(); }
```

### Joining Tables:

- How rows of data elements that are placed in tables are related to each other by linking rows using a common value in each row of two tables.
- Linking rows is called joining tables.
- Tables are joined in a query using a two-step process.



## Mobile Application Development Notes

- First, both tables that are being joined must be identified in the FROM clause, where tables are listed one after the other and are separated by a comma.
- Next, an expression is created in the WHERE clause that identifies the columns used to create the join.
- Let's say that an Orders table is joined to the Customers table using the customer number.
- The customer number in the Orders table is in the CustomerNumber column, and the customer number in the Customers table is the CustNum column.
  - ❖ WHERE CustomerNumber = CustNum
- The joined tables create a logical table that has all the columns of both tables.
- All the tasks performed on a single table can also be applied to joined tables.
- Joining too many tables can cause performance degradation and bring response time to a crawl. Typically, five is the maximum number of tables joined. However, the actual number may vary depending on the DBMS,

CustNumber	FirstName	LastName	Street	Zip
591	Anne	Smith	65 Cutter Street	04735
721	Bart	Adams	15 W. Spruce	05213
845	Tom	Jones	35 Pine Street	07660
901	Mary	Smith	5 Maple Street	08513

**Table 11-5.** Sample Customers Table for Practice Joining Tables

ZipCode	City	State
04735	Woodridge	TX
05213	River Ville	CA
07660	West Town	NJ
08513	SunnySide	NY

**Table 11-6.** Sample ZipCode Table for Practice Joining Tables

StoreNumber	ZipCode
278	08513
345	07660
547	05213
825	04735

**Table 11-7.** Sample Store Table for Practice Joining Tables

ProductNumber	ProductName	Unit Price
1052	CD Player	100
3255	VCR	250
5237	DVD Player	325
7466	50-inch TV	532

**Table 11-8.** Sample Products Table for Practice Joining Tables

OrderNumber	ProdNumber	CustomerNumber	StoreNumber	Quantity	SubTotal
122	5237	591	345	1	325
334	3255	901	278	1	250
365	3255	901	278	4	1000
534	7466	591	825	3	1596
587	5237	845	345	1	325
717	1052	721	825	2	200
874	7466	721	825	1	532

**Table 11-9.** Sample Orders Table for Practice Joining Tables

```
try { String query = new String ( "SELECT FirstName, LastName, City, State, ZipCode " +  
"FROM Customers, ZipCode " + "WHERE Zip = ZipCode");
```

```
DataRequest = Database.createStatement(); Results = DataRequest.executeQuery (query);  
DisplayResults (Results); DataRequest.close();
```

```
}
```

```
private void DownRow ( ResultSet DisplayResults ) throws SQLException { String FirstName=  
new String(); String LastName= new String(); String Street= new String(); String City = new  
String(); String State = new String(); String ZipCode= new String(); String printrow; FirstName  
= DisplayResults.getString ( 1 ) ; LastName = DisplayResults.getString ( 2 ) ; Street =  
DisplayResults.getString ( 3 ) ; City = DisplayResults.getString ( 4 ) ; State =  
DisplayResults.getString ( 5 ) ; ZipCode = DisplayResults.getString ( 6 ) ; printrow = FirstName  
+ " " + LastName + " " + City + " " + State + " " + ZipCode + " " + Sales + " " + Profit;  
System.out.println(printrow); }
```

### Parent-Child Join

- A parent-child join is used to join tables that have a parent-child relationship.
- This means rows in the parent table must exist for rows in the child table to exist.
- The products table and Orders table are joined using the product number value.
- Only rows that have a matching product number in both tables are placed in the virtual table that consists of rows from both tables.
- The Products table is the parent table in this relationship because an order cannot be placed without a product appearing in the Products table.

```
try { String query = new String ( " SELECT OrderNumber, ProductName " + " FROM Orders,  
Products " + " WHERE ProdNumber = ProductNumber"); DataRequest =  
Database.createStatement(); Results = DataRequest.executeQuery (query); DisplayResults  
(Results); DataRequest.close();
```

```
}
```

```
private void DownRow ( ResultSet DisplayResults ) throws SQLException { int OrderNum,  
ProdNum; String printrow = new String(); OrderNum = DisplayResults.getInt ( 1 ) ; ProdNum =
```

```
DisplayResults. getInt ( 2 ) ; printrow = OrderNum + " " + ProdNum;  
System.out.println(printrow); }
```

### Multiple Comparison Join

- The **WHERE** clause expression used to join tables can be a compound expression consisting of two or more sub expressions.
  - A compound expression is used to specify more than one selection criteria used to join two tables.
  - There are two components of the subexpression in this example. The first requires the DBMS to match product numbers in both tables. The other subexpression requires that the value in the Quantity column is greater than 2.
  - The **AND** operator is used to join both subexpressions. This means that both subexpressions must evaluate true for the DBMS to join rows of both tables. That is, only when the product number matches in both tables and the quantity of the order is greater than 2 will a row be included in the temporary table. If either subexpression evaluates false, the row is not included in the virtual table.
- ```
try { String query = " SELECT OrderNumber, ProductName, Quantity " + " FROM  
Orders, Products " + " WHERE ProdNumber = ProductNumber " + " AND Quantity > 2";  
DataRequest = Database.createStatement(); Results = DataRequest.executeQuery  
(query); DisplayResults (Results); DataRequest.close();
```

```
}
```

```
private void DownRow ( ResultSet DisplayResults ) throws SQLException { int OrderNum,  
ProdNum, Quantity; String printrow = new String(); OrderNum = DisplayResults.getInt ( 1 ) ;  
ProdNum = DisplayResults. getInt ( 2 ) ; Quantity = DisplayResults. getInt ( 3 ) ; printrow =  
OrderNum + " " + ProdNum + " " + Quantity; System.out.println(printrow); }
```

### Multitable Join :

- More than two tables can be joined together by using the name of each table in the join in the **FROM** clause and by defining the join with the appropriate column names in the **WHERE** clause expression.
- There are three tables joined in this example—Customers, Orders, and Products, as shown in the **WHERE** clause. Notice that all three tables don't need to have a common value used to join them. Each pair of tables has a value common to both of them. Customer numbers that are common between the tables join the Customers table and the Orders table. The Orders table and the Products table are joined by the product number.

```
try { String query = new String ( "SELECT FirstName, LastName,OrderNumber, ProductName,
Quantity " + " FROM Customers, Orders, Products " + " WHERE ProdNumber =
ProductNumber " + " AND CustNumber = CustomerNumber"); DataRequest =
Database.createStatement(); Results = DataRequest.executeQuery (query); DisplayResults
(Results); DataRequest.close();
}
```

```
private void DownRow ( ResultSet DisplayResults ) throws SQLException { int OrderNum,
Quantity; String FirstName = new String(); String LastName = new String(); String
ProductName = new String(); String printrow = new String(); FirstName =
DisplayResults.getString ( 1 ) ; LastName = DisplayResults.getString ( 2 ) ; OrderNum =
DisplayResults.getInt ( 3 ) ; ProductName = DisplayResults.getString ( 4 ) ; Quantity =
DisplayResults. getInt ( 5 ) ; printrow = FirstName + " " + LastName + " " + OrderNum + " " +
ProductName + " " + Quantity ; System.out.println(printrow); }
```

### Create a Column Name Qualifier

- Column names should reflect the kind of data element stored in the column, there is likely to be a conflict when two or more tables use the same column name to store the data element.
- This is the case when both the Products table and the Orders table contain product numbers.
- Conflict of column names can be resolved in a join by using a column name qualifier, which identifies the table that contains the column name.

```
try{ String query = new String ("SELECT Customers.CustNumber, " + " FirstName,
LastName, OrderNumber, " + " ProductName, Quantity " + " FROM Customers, Orders,
Products " + " WHERE ProdNumber = ProductNumber " + " AND
Customers.CustomerNumber = Orders.CustomerNumber"); DataRequest =
Database.createStatement(); Results = DataRequest.executeQuery (query);
DisplayResults (Results); DataRequest.close(); }
```

```
private void DownRow ( ResultSet DisplayResults ) throws SQLException { int
CustomerNumber, OrderNum, Quantity; String FirstName = new String(); String LastName =
new String(); String ProductName = new String(); String printrow = new String();
CustomerNumber = DisplayResults. getInt ( 1 ) ; FirstName = DisplayResults.getString ( 2 ) ;
LastName = DisplayResults.getString ( 3 ) ; OrderNum = DisplayResults.getInt ( 4 ) ;
ProductName = DisplayResults.getString ( 5 ) ; Quantity = DisplayResults. getInt ( 6 ) ; printrow
= CustomerNumber + " " + FirstName + " " + LastName + " " + OrderNum + " " + ProductName
+ " " + Quantity ; System.out.println(printrow); }
```

### Create a Table Alias:

- A query can be made readable by using table aliases.

- A table alias is an abbreviation for the name of the table that is used in place of the table name in the join and in the **SELECT** statement.
  - With as few letters as possible to save space in the query string
  - Representative of the table name, such as the first letter(s) of the name
  - Unique and not a duplicate of another table name, table alias, or column name

```
try { String query = new String ("SELECT c.CustNumber , " + " c.FirstName, c.LastName,
o.OrderNumber, " + " p.ProductName, o.Quantity " + " FROM Customers c, Orders o, Products
p" + " WHERE o.ProdNumber = p.ProductNumber " + " AND c.CustomerNumber =
o.CustomerNumber"); DataRequest = Database.createStatement(); Results =
DataRequest.executeQuery (query); DisplayResults (Results); DataRequest.close(); }
```

### Inner and Outer Joins :

- Link rows that have matching values in the column specified in the join.
- A row in a table that doesn't match is excluded from the join.
- There are two kinds of joins, each of which either excludes or includes rows in both tables of the join that don't match. These are
  - An inner join excludes rows of either table that don't have a matching value.
  - An outer join includes rows of either table that don't have a matching value.
- some DBMSs may use their own syntax rather than the SQL syntax for some or all joins.
- Therefore, consult with your database administrator or DBMS manufacturer for the proper syntax for joins if examples in this book don't work with your DBMS.
- Code segments used in these sections are executed using the **executeQuery()** method.
- Therefore, you'll need to use the Model B program. Remember that joins can also be used with the **UPDATE** and **DELETE** statements, in which case the Model A program is used.

### Inner Join

- an inner join to include only rows of both tables that have matching values.
  - Unmatched rows are excluded, and therefore those rows are not returned in the ResultSet.
- ```
try { String query = new String (" SELECT FirstName, LastName, OrderNumber,
ProductName, Quantity " + " FROM Customers,Orders, Products " + " WHERE
ProdNumber = ProductNumber " + " AND Customers.CustNumber =
Orders.CustomerNumber"); DataRequest = Database.createStatement(); Results =
DataRequest.executeQuery (query); DisplayResults (Results); DataRequest.close(); }
```

```
private void DownRow ( ResultSet DisplayResults ) throws SQLException { int OrderNum,
Quantity; String FirstName = new String(); String LastName = new String(); String
ProductName = new String(); String printrow = new String(); FirstName =
DisplayResults.getString ( 1 ) ; LastName = DisplayResults.getString ( 2 ) ; OrderNum =
```

```
DisplayResults.getInt ( 3 ) ; ProductName = DisplayResults.getString ( 4 ) ; Quantity =  
DisplayResults. getInt ( 5 ) ; printrow = FirstName + " " + LastName + " " + OrderNum + " " +  
ProductName + " " + Quantity ; System.out.println(printrow); }
```

### Outer Join

- Left, Right, Full An outer join occurs when matching and nonmatching rows of either or both tables are contained in the join.
- There are three kinds of outer joins:

#### Left outer join

- All matched and unmatched rows of the first table and matched rows of the second table are included in the join.
- Right outer join Matched rows of the first table and matched and unmatched rows of the second table are included in the join.
- Full outer join Matched and unmatched rows of both tables are included in the join.

#### Left Outer Join

- A join between the Customers table and the Orders table using the customer number.
- The \*= operator is used in the **WHERE** clause to create a left outer join.
- Think of the asterisk as a wildcard that tells the DBMS to use any row in the first table.

```
try { String query = new String ( " SELECT FirstName, LastName,OrderNumber " + "  
FROM Customers LEFT JOIN Orders " + " ON Customers.CustNumber =  
Orders.CustomerNumber"); DataRequest = Database.createStatement(); Results =  
DataRequest.executeQuery (query); DisplayResults (Results); DataRequest.close(); }
```

```
private void DownRow ( ResultSet DisplayResults ) throws SQLException { int, OrderNum;  
String FirstName = new String(); String LastName = new String(); String printrow = new  
String(); FirstName = DisplayResults.getString ( 1 ) ; LastName = DisplayResults.getString ( 2 )  
;
```

```
OrderNum = DisplayResults.getInt ( 3 ) ; printrow = FirstName + " " + LastName + " " +  
OrderNum; System.out.println(printrow);
```

```
}
```

#### Right Outer Join

- All rows in the Orders table are used in the join regardless of whether the customer number in the Orders table has a corresponding customer number in the Customers table.
- The WHERE clause contains the right outer join operator (>=\*), which, as you can see, has the asterisk positioned to the right instead of the left of the equivalent operator.

```
try { String query = new String ( " SELECT FirstName, LastName,OrderNumber" + "  
FROM Customers c, Orders o" + " WHERE c.CustNumber =* o.CustomerNumber");  
DataRequest = Database.createStatement(); Results = DataRequest.executeQuery  
(query); DisplayResults (Results); DataRequest.close(); }
```

### Full Outer Join

- A full outer join uses all the rows of both tables regardless of whether they match.
- The full outer join operator (**\*=**) is a combination of the left and right outer join operators, although some DBMSs use **FULL JOIN** or other syntax to create a full join.
- Consult with your database administrator or the DBMS manufacturer for the syntax used by your DBMS.

```
try { String query = new String( " SELECT FirstName, LastName,OrderNumber " + " FROM  
Customers c, Orders o" + " WHERE c.CustNumber *=* o.CustomerNumber"); DataRequest =  
Database.createStatement(); Results = DataRequest.executeQuery (query); DisplayResults  
(Results); DataRequest.close(); }
```

### CALCULATING DATA

- The DBMS can calculate values in a table and return the result of the calculation in the ResultSet by using one of the five built-in calculation functions:
  - **SUM()** tallies values in a column passed to the built-in function.
  - **AVG()** averages values in a column passed to the built-in function.
  - **MIN()** determines the minimum value in a column passed to the built-in function.
  - **MAX()** determines the maximum value in a column passed to the built-in function.
  - **COUNT()** determines the number of rows in a column passed to the built-in function.

### SUM()

- The **SUM()** built-in function calculates the sum of the values in the column that is passed to the function.
- values in the Quantity column in the Orders table are passed to the SUM() built-in function.

```
try { String query = new String ("SELECT SUM(Quantity) " + "FROM Orders ");  
DataRequest = Database.createStatement(); Results = DataRequest.executeQuery  
(query); DisplayResults (Results);
```

```
DataRequest.close();
```

```
}
```



```
private void DownRow ( ResultSet DisplayResults ) throws SQLException { long ReturnValue;
ReturnValue = DisplayResults.getLong ( 1 ) ; System.out.println(ReturnValue); }
```

### AVG()

- The **AVG()** built-in function calculates the average value in the column passed to the function. The average is returned in the ResultSet.
- The **AVG()** built-in function to calculate the average of the values in the Quantity column of the Orders table.  
try { String query = new String ("SELECT AVG(Quantity) " + "FROM Orders ");  
DataRequest = Database.createStatement(); Results = DataRequest.executeQuery  
(query); DisplayResults (Results); DataRequest.close(); }

### MIN()

- The **MIN()** built-in function returns the lowest value contained in the column passed to the function.
- The **MIN()** built-in function to determine the minimum value in the Quantity column of the Orders table.

```
try { String query = new String ("SELECT MIN(Quantity) " + "FROM Orders ");  
DataRequest = Database.createStatement(); Results = DataRequest.executeQuery  
(query); DisplayResults (Results); DataRequest.close(); }
```

### MAX()

- The **MAX()** built-in function returns the highest value contained in the column passed to the built-in function.
- The **MAX()** built-in function to determine the maximum value in the Quantity column of the Orders table.

```
try { String query = new String ("SELECT MAX(Quantity) " + "FROM Orders "); DataRequest  
= Database.createStatement(); Results = DataRequest.executeQuery (query); DisplayResults  
(Results); DataRequest.close(); }
```

### COUNT()

- Counting the number of rows or values in a table is a common calculation.
- The **COUNT()** built-in function returns the number of rows in a column.
- The **COUNT()** built-in function to determine the number of values that appear in the Quantity column of the Orders table.
- The result is returned in the ResultSet. Rows without values in the column are excluded from the count.
- This means the value returned does not necessarily represent the total number of rows in the table.



```
try { String query = new String ("SELECT COUNT(Quantity) " + "FROM Orders ");  
DataRequest = Database.createStatement(); Results = DataRequest.executeQuery (query);  
DisplayResults (Results); DataRequest.close(); }
```

### Count All Rows in a Table

- The **COUNT()** built-in function is also used to return the number of rows in a table.
- This is accomplished by passing the **COUNT()** built-in function an asterisk rather than the name of a column.

```
try { String query = new String ("SELECT COUNT(*) " + "FROM Orders ");  
DataRequest = Database.createStatement(); Results = DataRequest.executeQuery  
(query); DisplayResults (Results); DataRequest.close(); }
```

### Retrieve Multiple Counts :

- Multiple counts can be returned in the ResultSet by using more than one **COUNT()** built-in function in the **SELECT** statement.
- Where the ResultSet contains the total number of rows in the Orders table and the number of rows from the Orders table that contain values in the Quantity column.

```
try { String query = new String ("SELECT COUNT(*), COUNT(Quantity) " + "FROM  
Orders "); DataRequest = Database.createStatement(); Results =  
DataRequest.executeQuery (query); DisplayResults (Results); DataRequest.close(); }
```

```
private void DownRow ( ResultSet DisplayResults ) throws SQLException { long TotalRows,  
TotalValues; TotalRows = DisplayResults.getLong ( 1 ); TotalValues = DisplayResults.getLong  
( 2 ); System.out.println( "Rows: " + TotalRows + "\nValues: " +TotalValues);  
}
```

### Calculate a Subset of Rows

- Can restrict the scope of a built-in calculation function by using a **WHERE** clause expression to specify the criteria for a row to be included in a calculation.
- Any valid **WHERE** clause expression can be used to filter rows to be excluded from the calculation.
- Likewise, the **WHERE** clause expression is used to include rows in the calculation.
- How to restrict the scope of a built-in calculation function by using a **WHERE** clause expression.
- The **DBMS** is told to count the number of values in the OrderNumber column of the Orders table and to average and total the value of the Quantity column of the Orders table.
- The **WHERE** clause contains an expression that joins the Customers table and the Orders table using the customer number as the common value between these tables.
- This means the calculations are performed only on rows that are in the join.

- Rows whose customer numbers don't appear in both the Customers table and Orders table are excluded from the calculations.

```
try { String query = new String ( " SELECT COUNT(OrderNumber), AVG(Quantity),  
SUM(Quantity) " + " FROM Orders o, customers c " + " WHERE o.CustomerNumber =  
c.CustNumber"); DataRequest = Database.createStatement(); Results =  
DataRequest.executeQuery (query); DisplayResults (Results); DataRequest.close(); }
```

```
private void DownRow ( ResultSet DisplayResults ) throws SQLException { long Orders; long  
Average; long Sum; Orders = DisplayResults.getLong ( 1 ); Average = DisplayResults.getLong  
( 2 ); Sum = DisplayResults.getLong ( 3 ); System.out.println("Total Orders = " + Orders);  
System.out.println("Average Qty = " + Average); System.out.println("Total Qty = " + Sum);  
}
```

### NULLs and Duplicates:

- Two common problems that occur when using built-in functions are columns that don't contain a value and rows that contain duplicate values in the same column.
- Manytimes you don't want empty columns and duplicate rows included in the calculation.
- We can use the DISTINCT modifier to exclude duplicate rows from the calculation. Likewise, problems posed by NULLcolumns can be avoided by using the IS NULL operator along with the NOT operator in a selection expression.

### Calculate Without Using Built-in Functions

- Although built-in calculation functions are very useful,the DBMS can perform calculations that are defined in the **SELECT** statement.
- The DBMS is directed to return to the program the value of the StoreNumber column and the difference between the value in the Sales column and the value in the Estimate column.
- Neither the Sales column nor the Estimate column data is returned. Instead, only the difference between these two columns is returned.
- Any arithmetic expression can be used in the **SELECT** statement along with the appropriate names of columns that contain data elements used in the calculation.

```
try { String query = new String ( " SELECT StoreNumber, Sales - Estimate " + " FROM  
Sales "); DataRequest = Database.createStatement(); Results =  
DataRequest.executeQuery (query); DisplayResults (Results); DataRequest.close(); }
```

```
private void DownRow ( ResultSet DisplayResults ) throws SQLException { String Store; long  
Difference; Store = DisplayResults.getString ( 1 ); Difference = DisplayResults.getLong ( 2 );  
System.out.println("Store = " + Store); System.out.println("Difference = " + Difference);
```

}

### Grouping and Ordering Data

- Columns are returned in the ResultSet in the order that the column names appear in the statement of the query.
- The order in which rows appear in the ResultSet can be grouped into similar values or sorted in ascending or descending order by using the **GROUPBY** clause or the **ORDERBY** clause.
- Grouping is the task of organizing rows of data according to similar values within the same column. Let's say that you want to see sales for each store.
- The ResultSet can be grouped by store number.  
Sorting is the task of organizing rows of data in either alphabetical or numerical order according to the value of a column in the result set.
- ADBMS is capable of creating simple and complex sorting. A simple sort is when the values in a single column are used for the sort.
- A complex sort is when multiple columns are used for the sort, such as sorting rows by last name and within last names, by first names.

### GROUP BY

- The **GROUPBY** clause specifies the name of the column whose values are used to group rows in the ResultSet.
- The StoreNumber column and the sum of the values in the Sales column from the Sales table are returned.
- The **GROUP BY** clause organizes the ResultSet by the value in the StoreNumber column.

```
try { String query = new String (" SELECT StoreNumber, SUM(Sales) " + " FROM  
Sales " + " Group By StoreNumber"); DataRequest = Database.createStatement();  
Results = DataRequest.executeQuery (query); System.out.println("Store    Sales");  
System.out.println("-----    -----"); DisplayResults (Results); DataRequest.close(); }
```

```
private void DownRow ( ResultSet DisplayResults ) throws SQLException { long Store; long  
Sum; Store = DisplayResults.getLong ( 1 ) ; Sum = DisplayResults.getLong ( 2 ) ;  
System.out.println(Store + "          " + Sum); }
```

### Group Multiple Columns

- The DBMS can create a subgroup within a group in the ResultSet.
- A sub group is created by placing the name of the column used for the sub group as the second column name in the **GROUP BY** clause of the query.
- Any number of subgroups can be created, depending on the limitations established by the manufacturer of the DBMS used by your program.

- Column names in the **GROUP BY** clause must be separated with a comma.

```
try { String query = new String ( " SELECT StoreNumber,SalesRepNumber, SUM(Sales)
" + " FROM Sales " + " Group By StoreNumber, SalesRepNumber"); DataRequest =
Database.createStatement(); Results = DataRequest.executeQuery (query);
System.out.println("Store SalesRep Sales"); System.out.println("-----");
DisplayResults (Results); DataRequest.close(); }
```

```
private void DownRow ( ResultSet DisplayResults ) throws SQLException { long
StoreNumber; long SalesRepNumber; long Sum; StoreNumber = DisplayResults.getLong ( 1 );
SalesRepNumber = DisplayResults.getLong ( 2 ); Sum = DisplayResults.getLong ( 3 );
System.out.println( StoreNumber + " " + SalesRepNumber + " " + Sum); }
```

### Conditional Grouping

- The number of rows that are included in a group can be limited by a conditional expression in the query.
- A conditional expression is similar to the WHERE clause expression.
- The DBMS uses the conditional expression to qualify whether or not the current row should be included in any group of the ResultSet.
- Only rows that meet the condition are returned. A row that doesn't meet the condition is excluded.
- The conditional expression is placed in the HAVING clause of the query.
- The HAVING clause sets the criteria for a row to be included in a group.
- The value of the StoreNumber column and the total sales for each store from the Sales table are grouped by store number.
- The HAVING clause excludes from the group, stores that have total sales of less than \$401

```
try { String query = new String ("SELECT StoreNumber, SUM(Sales) " + " FROM Sales
" + " Group By StoreNumber" + " HAVING SUM(Sales) > 400"); DataRequest =
Database.createStatement(); Results = DataRequest.executeQuery (query);
System.out.println("Store Sales"); System.out.println("-----"); DisplayResults
(Results); DataRequest.close(); }
```

```
private void DownRow ( ResultSet DisplayResults ) throws SQLException { long
StoreNumber; long Sum; StoreNumber = DisplayResults.getLong ( 1 ); Sum =
DisplayResults.getLong ( 2 ); System.out.println(StoreNumber + " " + Sum); }
```

### Working with NULL Columns

- Empty columns can create unexpected results when you execute a query.

- This is because, depending on the nature of the query, the empty column may be included or excluded from the operation.
- The DBMS includes empty columns when calculating average values and counting rows, but excludes empty columns when calculating the minimum or maximum value within a column.
- The DBMS may include or exclude a row in a group depending on the conditional expression.
  - A row is included in a group if the empty column isn't used to group rows or used in the conditional expression in the HAVING clause.
  - A row is excluded from the group if the empty column is used in the conditional expression and the conditional expression normally excludes empty columns from the calculation (as with MIN(), MAX()).
  - A row is included in the group if the empty column is used in the conditional expression and the conditional expression normally includes empty columns from the calculation (as with AVG(), COUNT()).
  - A row is included in the group if the empty column is used to group rows. Rows containing the empty column are placed in their own group. Sorting Data The ResultSet can be placed in alphabetical or numerical order by using the ORDER BY clause in the query.

```
try { String query = new String ("SELECT StoreNumber, Sales " + " FROM Sales " + " ORDER  
BY StoreNumber"); DataRequest = Database.createStatement(); Results =  
DataRequest.executeQuery (query); System.out.println("Store Sales"); System.out.println("-----  
-----"); DisplayResults (Results); DataRequest.close(); }
```

```
private void DownRow ( ResultSet DisplayResults ) throws SQLException { long  
StoreNumber; long Sales; StoreNumber = DisplayResults.getLong ( 1 ) ; Sales =  
DisplayResults.getLong ( 2 ) ; System.out.println(StoreNumber + " " + Sales); }
```

### Major and Minor Sort Keys

- Can create a sort with in a sort by specifying more than one column to be sorted, such as sorting rows by customer last name, and then within last name, sorting by customer firstname.
- The first column specified in the **ORDER BY** clause is called the major sort key and is the initial value used to sort rows.
- The second and subsequent columns in the **ORDER BY** clause are called minor sort keys. A comma must separate each column name.
- The StoreNumber column and Sales column from the Salestable are returned in the ResultSet.
- The **ORDER BY** clause sorts the ResultSet by values in the StoreNumber column. Within each StoreNumber value, the value in the Sales column is sorted. Both sorts are ascending numerical sorts.

```
try { String query = new String ("SELECT StoreNumber, Sales " + " FROM Sales " + "
ORDER BY StoreNumber, Sales"); DataRequest = Database.createStatement(); Results =
DataRequest.executeQuery (query); System.out.println("Store Sales");
System.out.println("----- -----"); DisplayResults (Results); DataRequest.close(); }
```

### Descending Sort

- In addition to choosing the column to sort, you can also select the direction of the sort by using the ASC or DESC modifier.
- The DESC modifier in the ORDER BY clause to specify a descending sort.
- By default the sort is ascending, although you can use the ASC modifier to explicitly direct that the sort be in ascending order.
- The ASC or DESC modifier must appear after the column name in the ORDER BY clause.
- The ASC or DESC modifier can be used for major and minor sort keys.

```
try { String query = new String ("SELECT StoreNumber, Sales " " FROM Sales " + "
ORDER BY StoreNumber DESC "); DataRequest = Database.createStatement(); Results
= DataRequest.executeQuery (query); System.out.println("Store Sales");
System.out.println("----- -----"); DisplayResults (Results); DataRequest.close(); }
```

### Sorting on Derived Data

- Data that doesn't exist in a table, but comes from the data in a table, such as a calculation is derived data.
- Placing a calculation in the **SELECT** statement creates derived data.
- The store number and the difference between the value in the Sales column and the value in the Estimate column from the Sales table. The difference is derived data.
- Derived data doesn't have a column name. This means that you cannot include a column name in the **ORDER BY** clause to designate the derived data as the major or minor sort key for the sort.
- Can use the column number of derived data in place of a column name to sort the derived data.
- The column number of derived data corresponds to the position of the derived data in the **SELECT** statement.
- The calculation expression that produces the derived data is the second data referenced in the **SELECT** statement.
- Therefore, the derived data appears in the second column of the ResultSet. Use 2 in place of the column name in the **ORDER BY** clause.

```
try { String query = new String ( " SELECT StoreNumber, (Sales-Estimate) " + " FROM  
Sales " + " ORDER BY 2 "); DataRequest = Database.createStatement(); Results =  
DataRequest.executeQuery (query); System.out.println("Store Sales");  
  
System.out.println("----- -----"); DisplayResults (Results); DataRequest.close();  
  
}
```

### Subqueries

- In the real world, will find that you need to create complex queries that do more than simply request columns from the database. Instead, you might request columns based on the result of another query.
- We can direct the DBMS to query the result of a query by creating a subquery. A subquery joins two queries to form one complex query, which efficiently identifies data to be included in the ResultSet.
- The format of a subquery is similar to a query, but rows that are selected as a result of the subquery are not returned to your program.
- Instead, selected rows are queried by another query. Rows chosen by the second query are returned in the ResultSet.
- Both queries and subqueries have a SELECT statement and a FROM clause and can also include a WHERE clause and a HAVING clause to qualify rows to return. The WHERE clause and HAVING clause are used to express a condition that must be met for a row to be included in the result set.
- There are also differences between a query and a subquery. The most noticeable difference is with the ResultSet.
- The ResultSet of a query is returned to the J2ME component. In contrast, the result set of a subquery is returned to a temporary table, which is then used by a query or another subquery to further extract rows.
- This means the J2ME application never sees the result of a subquery. Instead, the subquery result is an intermediate step working toward the final selection of data from the database. You must follow two rules when using a subquery in your program:
  - Return one column from the subquery. The purpose of a subquery is to derive a list of information from which a query can choose appropriate rows. Only a single column needs to be included in the list.
  - Don't sort or group the result from a subquery. Since the ResultSet of the subquery isn't going to be returned in the ResultSet of the query, there is no need to sort or group data in the ResultSet of a subquery.

### Create a Subquery



- A subquery is a query whose results are evaluated by an expression in the WHERE clause of another query.
- The subquery is defined below the first WHERE clause.
- The subquery joins rows of the Sales table and the Order subquery totals the value of the Amount column for each store number and returns the results to a temporary table.
- The query then returns a ResultSet that contains store numbers where the value of the Estimate column in the Sales table is equal to the total number of the Amount column returned by the subquery.

```
try { String query = new String (" SELECT StoreNumber " + " FROM Sales " + " WHERE Estimate = (SELECT SUM(Amount) " + " FROM Orders, Sales " + " WHERE StoreNum = StoreNumber) "); DataRequest = Database.createStatement(); Results = DataRequest.executeQuery (query); System.out.println("Store"); System.out.println("-----"); DisplayResults (Results); DataRequest.close(); }
```

```
private void DownRow ( ResultSet DisplayResults ) throws SQLException { long Store; Store = DisplayResults.getLong ( 1 ) ; System.out.println(Store); }
```

### Conditional Testing

- Any conditional expression can be used to evaluate a relationship between a query and the results of the subquery.
- Can use four types of conditional tests with a subquery:
  - **Comparison test** This test uses comparison operators to compare values in the temporary table with values in the table used by the query.
  - **Existence test** This test determines whether a value in the current row of the table used by the query also exists in the temporary table.
  - **Set membership test** This test is similar to the existence test in that the DBMS is directed to determine whether a value in the current row of the table used by the query also exists in the temporary table.
  - **Qualified test** This test consists of either the ANY test or the ALL test and determines whether a value in the current row of the table used by the query is in one row of the temporary table or all rows of the temporary table.
- **Existence Test** The existence test is used whenever you need to return rows in the ResultSet where a value in a column is present in the results of the subquery.
- The **existence test** requires that you place the EXISTS modifier between the query and the subquery.

```
try { String query = new String (" SELECT DISTINCT StoreNumber " + " FROM Sales " + " WHERE EXISTS " + " (SELECT StoreNum " + " FROM Orders " + " WHERE StoreNum = StoreNumber) "); DataRequest = Database.createStatement(); Results =
```



```
DataRequest.executeQuery (query); System.out.println("Store"); System.out.println("-----"); DisplayResults (Results); DataRequest.close(); }
```

- **Membership Test** The **IN** modifier is used to determine whether a value in the table that is being queried is a member of the results produced by the subquery.
- If the value appears in both the table and the results of the subquery, the conditional test is evaluated as true and the row that contains the value is returned in the **ResultSet**; otherwise the row is skipped.
- The **IN** modifier is used in the subquery returns store numbers from the **Orders** table where the value in the **Estimate** column of the corresponding row in the **Sales** table is less than the value in the **Amount** column of the **Orders** table.
- The **IN** modifier is used to determine whether the store number in the **Sales** table is returned in the results of the subquery. If so, then the sales representative's number from the **Sales** table is returned in the **ResultSet**; if not, the sales representative's number in the current row of the **Sales** table is skipped  

```
try { String query = new String (" SELECT SalesRepNumber " + " FROM Sales " + " WHERE StoreNumber IN " + " (SELECT StoreNum " + " FROM Orders " + " WHERE Estimate < Amount) "); DataRequest = Database.createStatement(); Results = DataRequest.executeQuery (query); System.out.println("Store"); System.out.println("-----"); DisplayResults (Results); DataRequest.close(); }
```
- **ANY Test** The **ANY** test determines whether the value specified in the query is in any of the rows returned by the subquery. If there is at least one match, the row is returned in the **ResultSet**, otherwise the row is ignored. shows how to construct an **ANY** test. The objective of this program is to return the store number from the **Sales** table for any store where the estimate is greater than the amount of any of the store's orders in the **Orders** table.
- The **ANY** test tells the DBMS to include the store number in the **ResultSet** if the value of the **Estimate** column for the store in the **Sales** table is greater than any value in the **Amount** column for that store in the temporary table.
- **Rules for the ANY Test** You must conform to a set of rules that govern the use of the **ANY** test; otherwise, you might experience unexpected results. Here are the rules that you must follow when using the **ANY** test:

- The **ANY** test fails if the subquery produces an empty result. Therefore, the query doesn't return any rows in the **ResultSet**.

- A **NULL** value returned by the subquery causes the query to return a **NULL** value. This is because the query is unsure whether the value of the table in the query is in the results of the subquery.

- No value is returned in the **ResultSet** if the value in the table being queried is not in the results of the subquery.

```
try { String query = new String (" SELECT DISTINCT StoreNumber " + " FROM Sales " + " WHERE Estimate > ANY " + " (SELECT Amount" + " FROM Orders " + " WHERE StoreNumber = StoreNum) "); DataRequest = Database.createStatement(); Results = DataRequest.executeQuery (query); System.out.println("Store"); System.out.println("-----"); DisplayResults (Results); DataRequest.close(); }
```

- **ALL Test** The **ALL** test is similar to the **ANY** test in that the value of the table specified in the query is compared to values returned by the subquery.
- However, these tests differ in that the **ANY** test requires at least one match for the row to be included in the **ResultSet**.
- The **ALL** test requires that the value in the table specified in the query matches all the values returned by the subquery.
- Once the tables are joined, the subquery is processed, which returns the value in the **Amount** column for each store in the **Orders** table.
- Next, the value in the **Estimate** column of the **Sales** table is compared with each **Amount** value returned by the subquery.
- If the **Amount** value for each row returned by the subquery is less than the value in the **Estimate** column, then the row in the **Sales** table is placed in the **ResultSet**. Otherwise, the row in the **Sales** table is skipped.

```
try { String query = new String (" SELECT DISTINCT Store " + " FROM Sales " + " WHERE Estimate > ALL " + " (SELECT Amount" + " FROM Orders " + " WHERE StoreNumber = StoreNum) "); DataRequest = Database.createStatement(); Results = DataRequest.executeQuery (query); System.out.println("Store"); System.out.println("-----"); DisplayResults (Results); DataRequest.close(); }
```

### VIEWS

- can reduce the complexity of your **J2ME** application by creating one or more views of the database for each user ID that is passed to the **J2ME** application for data access.
- **A VIEW** is similar to creating a table that contains only data the user ID is permitted to access.
- **A VIEW** limits columns and rows that can be queried to specific information pertaining to a user ID.
- **A VIEW** is like a filter that hides information from a user ID. Each **VIEW** is uniquely identified with a name and contains selection criteria for columns and rows that appear in the **VIEW** when the **VIEW** is used by a **J2ME** component.
- Once a **VIEW** is created, the **J2ME** application references a **VIEW** the same way that a table is referenced in a query.
- **Rules for Using VIEWS** In many situations, using **VIEWS** increases the efficiency of interacting with tables in a database.
- There are times when **VIEWS** are not appropriate for an application.

- The following is a set of rules that govern how you should use VIEWS in your application:
  - Create as many VIEWS as necessary to simplify access to a database.
  - Restrict access to a table on need-to-know basis.
  - Work with the owner of the data to establish reasonable restrictions.
    - Classify users into groups that have similar access requirements to information.
  - Create a VIEW for each classification of user rather than for each user.
  - More than one column can be used in a VIEW.
  - More than one table can be used in a VIEW.
  - A VIEW is treated as a table in a query regardless of the number of columns and tables that are used to create the VIEW.
  - Use a VIEW whenever your program accesses some columns in many tables. The view simplifies the number of tables that a J2ME application needs to access directly.
    - Beware of data security restrictions that affect the underlying columns and tables used to create the VIEW. A VIEW inherits data security restrictions from tables used to create the VIEW. Therefore, if the user ID doesn't have rights to information in a table, the VIEW also has the same restrictions.
  - Create as many VIEWS as necessary to simplify access to a database. However, querying a VIEW is not necessarily an efficient way to query a database.

### Create a VIEW

- A VIEW is created by using the CREATE VIEW statement, the CREATE VIEW statement contains the name of the VIEW.
- Any unique name can be used as the name of a VIEW. The AS modifier contains the query whose results form the rows and columns that are contained in the VIEW.

```
try { String query = new String (" CREATE VIEW Store278 AS " + " SELECT  * " + "
FROM Orders  " + " WHERE StoreNum = 278"); DataRequest =
Database.createStatement(); DataRequest.execute(query); DataRequest.close(); }
```

- Select Columns to Appear in the VIEW You can include or exclude any column in a VIEW.
- Columns excluded from a VIEW remain in underlying tables used to create the VIEW.
- The SELECT statement contains column names from the Order table that are included in the view.
- All other columns of the Orders table are excluded from the view.

```
try { String query = new String (" CREATE VIEW StoreProd AS " + " SELECT StoreNum,  
ProdNumber " + " FROM Orders "); DataRequest = Database.createStatement();  
DataRequest.executeQuery (query); DataRequest.close(); }
```

### Create a Horizontal VIEW

- There are two kinds of VIEWS:
  - ❖ **Vertical and horizontal.**
- A **verticalVIEW** includes all rows of the underlying table and includes some, but not all, columns of the table.
- A **horizontal VIEW** contains all columns in the underlying table, but only some rows of the table. This means some rows are excluded from the VIEW and cannot be accessed.
- **Horizontal VIEWS** are ideal for situations where access to all columns of a table(s) is required, but not access to all rows.

```
try { String query = new String (" CREATE VIEW cust901 AS " + " SELECT * " + " FROM  
Orders" + " WHERE CustomerNumber = 901"); DataRequest = Database.createStatement();  
Results = DataRequest.execute(query); DataRequest.close(); }
```

### Create a Multitable VIEW

- A **multitable VIEW** is created like a single table VIEW, but tables used in the view must be joined.
- A **vertical VIEW** is created and contains three columns from the Orders table and the Products table.
- The Orders table and the Products table are joined together in the **WHERE** clause using the product number.
- Although columns in the view come from two tables, columns are treated as if they are from the same table in a query that uses the view.

```
try { String query = new String (" CREATE VIEW ProdDesc AS " + " SELECT StoreNum,  
ProdNumber, ProductName " + " FROM Orders, Products " + " WHERE ProdNumber =  
ProductNumber"); DataRequest = Database.createStatement(); Results =  
DataRequest.execute(query); DataRequest.close(); }
```

### Group and Sort VIEWS

- Rows in a **VIEW** can be grouped and sorted when the VIEW is created rather than grouping or sorting rows in a query that uses the VIEW.
- A **VIEW** can be grouped or sorted by using the **GROUP BY** clause and/or the **ORDER BY** clause.

- A **VIEW** called **GroupProdDesc** is created based on the underlying Orders table and Products table, which are joined together by product number.
- The **GroupProdDesc** view consists of the store number, product number, and product name that are sorted by the value in the **ProdNumber** column.

```
try { String query = new String (" CREATE VIEW GroupProdDesc AS " + " SELECT  
StoreNum, ProdNumber, ProductName " + " FROM Orders, Products " + " WHERE  
ProdNumber = ProductNumber" + " ORDER BY ProdNumber "); DataRequest =  
Database.createStatement(); Results = DataRequest.execute(query); DataRequest.close(); }
```

### Modify a VIEW

- A VIEW can be modified, and those modifications affect the underlying tables that are used to create the VIEW.
  - There are three ways to modify a VIEW:
    - Update Values in one or more columns of a VIEW are changed to values supplied by the query.
    - Insert Anew row is added to the VIEW and indirectly to the underlying table(s) used to create the VIEW.
    - Delete Arow is removed from the VIEW and from the underlying table(s) used to create the VIEW. Rules for Updating a View AView can be used to update columns and rows in the underlying tables that comprise the View. updating a View is possible only if you adhere to a set of rules that govern the use of updating a View.
    - Calculation, expressions, and built-in column functions cannot be used in the SELECT statement of the VIEW. Instead, only names of columns can be used.
    - Exclude the GROUPBYclause and HAVING clause from the VIEW.
    - Duplicate rows must be included in the modification. You cannot use the DISTINCT modifier in the VIEW if values of the view are going to be modified.
    - The user ID used in your program must have rights to modify the underlying tables that make up the VIEW.
    - Subqueries cannot be used in a VIEW if rows of the VIEW are to be modified.
- Updating a View You can replace values in a view by using the UPDATE statement, as discussed earlier in the section "Update Row and Column." The SET clause specifies the Amount column, which is to be updated, and the value that will override the current value in the Amount column.

```
try { String query = new String (" UPDATE Store278 " + " SET Amount = 700 " + " WHERE  
OrderNumber = 334 "); DataRequest = Database.createStatement(); Results =  
DataRequest.execute(query); DataRequest.close(); }
```

}

### Inserting a Row into a VIEW

- A new row can be inserted into the underlying tables that make up aVIEW by using the **INSERT INTO** statement and referencing the name of theVIEW,  
try { String query = new String (" INSERT INTO Store278 " + " (OrderNumber, ProdNum, CustomerNumber, StoreNum, Amount) " + " VALUES (325, 9545 ,301 ,278 ,400) "); DataRequest = Database.createStatement(); Results = DataRequest.execute(query); DataRequest.close(); }

### Deleting a Row from a VIEW

- Can remove a row from a VIEW by using the **DELETE FROM** statement.
- The **DELETE FROM** statement requires the name of the VIEW from which the row or rows is to be deleted.
- Rows designated to be deleted are identified in the **WHERE** clause.  
try { String query = new String (" DELETE FROM Store278 " + " WHERE OrderNumber = 555 "); DataRequest = Database.createStatement(); Results = DataRequest.execute(query); DataRequest.close(); }

### Dropping a VIEW

- A**VIEW** can be removed by using the **DROPVIEW** statement.
- The **DROPVIEW** statement requires the name of the VIEW that is to be dropped.
- Two modifiers are used with the **DROPVIEW** statement:
  - **CASCADE** Remove all **VIEW**s that depend on the **VIEW** specified in the **DROPVIEW** statement as well as the specified **VIEW**.
  - **RESTRICT** Remove only the **VIEW** specified in the **DROPVIEW** statement. All dependent **VIEW**s remain intact.
- **Adependent VIEW** It is a **VIEW** that has another **VIEW**, rather than a table, as one of itsunderlyingcomponents.  
try { String query = new String (" DROP VIEW Store278 CASCADE"); DataRequest = Database.createStatement(); Results = DataRequest.DataRequest.execute(query); DataRequest.close(); }

## GENERIC CONNECTION FRAMEWORK

---

- **The Connection:** A **connection** is a path between two computing devices that utilizes a hard-wired(cable) technology or wireless technology to transmit and receive data over a network.
- These devices can be a small computing device and a back-end processing system or two small computing devices.
- **Three pieces** of information are required to establish a connection between two computing devices.
- These are a **network address, communications protocol, and communication parameters.**
- The **network address** uniquely identifies each computing device on a network. An Internet protocol (IP) address is used on many networks, although other addressing schemes exist.
- A **communications protocol** is a set of rules that describe how data is transmitted between two computing devices. Both computing devices must agree on a communications protocol before transmitting data.
- **There are three widely used communications protocols:**
  - Hypertext Transfer Protocol (HTTP),
  - File Transfer Protocol (FTP), and
  - socket.
- **Communication parameters** consist of information required to open a connection between two computing devices.
- **Login information** such as a user ID and password are commonly used as communication parameters if the remote computer requires authentication before opening the connection with the other computing device. A
- Connection is opened by calling the **Connector.open()** method.
- The **Connector.open() method requires one parameter**, which is a String containing the communications protocol, network address, and any communication parameters.
- These components are separated within the String by a colon.
- Can disregard the last colon if no parameters are used to open the connection.
- The following code segment illustrates how to open a connection that uses HTTP as the communications protocol and //www.myweb.com as the network address. No communications are necessarily in this example.

```
Connection connection=Connector.Open("http://www.myweb.com");
```

- The **Class.forName()** method is automatically called at run time to determine the class that implements the protocol being used for the connection.
- The **Connector.open()** method returns an instance of the Connection Interface.
- **Connection and Streams :** The **Connector class** described in the GCF is used to establish network connections.



- The **Connector** class is used to access one of seven GCF connection interfaces.
- GCF connection interfaces provide a basic architecture for network operations and network protocol independence for writing network code.
- GCF connection interfaces are used the same way regardless of the underlying network protocol.
- These interfaces are **Connection**, **ContentConnection**, **DatagramConnection**, **InputConnection**, **OutputConnection**, **StreamConnection**, and **StreamConnectionNotifier**.
- **All of these are located in the `javax.microedition.io` package.**
- The **Connection interface** is the basic connection that can either open or close a connection.
- The **ContentConnection interface** is used in a streaming connection that accesses information located on a web server.
- The **DatagramConnection interface** is used for packet transmission through the use of a datagram connection.
- The **InputConnection interface** and **OutputConnection interface** are used to receive and send data from and to a communications device.
- The **StreamConnection interface** is used for two-way transmissions using a communications device.
- The **StreamConnection Notifier interface** is used when a stream connection is established.
- The **Connector** class and related connection interfaces only open (and close) a connection. They do not manage transmission.
- Think of the **Connector class** as the code that opens the telephone line for transmission to and from an Internet service provider.
- **Input/output classes defined in the `java.io` package are used to manage transmission over an open connection.**
- It is the input/output classes that write data to an open connection and read data from an open connection.
- The **InputStream** class and the **OutputStream** class are base classes for all input and output classes.
- **Derived from these classes are input and output classes that stream specific kinds of data.**
- **Derived classes are `ByteArrayInputStream`, `ByteArrayOutputStream`, `DataInputStream`, `DataOutputStream`, and `PrintStream`.**
- The **`ByteArrayInputStream` class and the `ByteArrayOutputStream` class** buffer internal byte arrays for input and output.
- The **`DataInputStream` class and the `DataOutputStream` class** are used to input and output data as primitive Java datatypes.



- The `PrintStream` class outputs primitive data individually. Characters are written and read by using the `InputStreamReader` class and the `OutputStreamWriter` class, which are derived from both the `InputStream` class and `Reader` class and the `OutputStream` class and `Writer` class, respectively.

### **InputStream Class:**

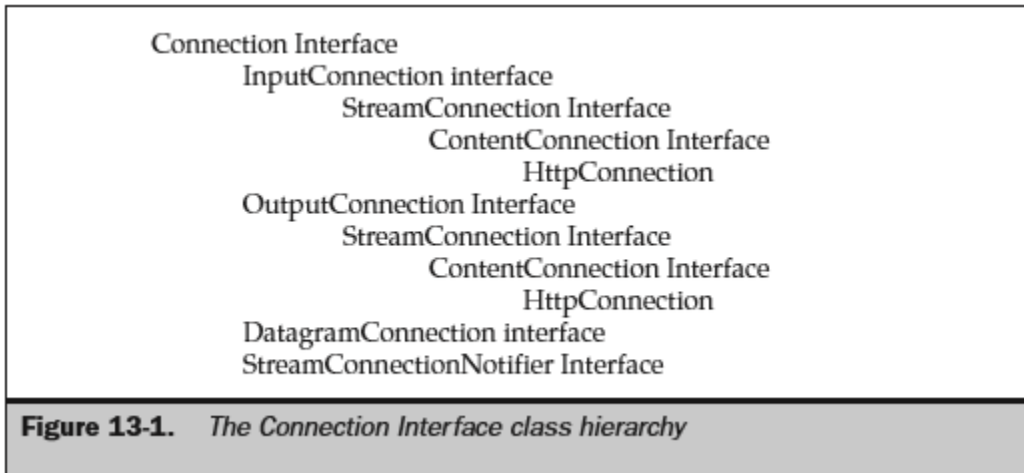
- The **`InputStream`** class defines the basic interface for reading data from an open connection.
- **There are nine methods in the `InputStream` class, including three versions of the `read()` method.**
  - These are `read()`, `read(byteb[])`, `read(byteb[],intoff,intlen)`,
  - `skip(longn)`,
  - `available()`,
  - `mark(intreadlimit)`,
  - `reset()`,
  - `markSupported()`, and
  - `close()`.
- The **first `read()` method** returns a byte of data as an integer from the open connection. If the end of the stream is reached, a `-1` is returned.
- The **second `read()` method** is used to read multiple bytes from the open connection, which are stored in an array passed as a parameter to the `read()` method. The return value of this version of the `read()` method is either the number of bytes read from the open connection or a `-1` indicating that the last byte has been read.
- The **third version of the `read()` method** also reads multiple bytes from an open connection to a byte array. we can specify where in the byte array to place bytes read from the connection. **This version requires three parameters.**
- **The first parameter is the array used to store incoming bytes.**
- **The second parameter is an `int` that specifies the first element of the array used to store the first incoming byte.**
- **The third parameter is an `int` that specifies the total number of bytes to read from the connection. This version of the `read()` method also returns either the total number of bytes read or a `-1`.**
- Can skip bytes in an input stream by calling the **`skip()` method**.
- The **`skip()` method requires one parameter, which is a `long` that specifies the number of bytes to skip.**
- The **`skip()` method** returns either the total number of bytes skipped or a `-1` indicating the end of the stream.
- The **`InputStream` class uses a blocking technique** when ever data is unavailable from the open connection.
- The **blocking technique requires the `InputStream` class to wait until data becomes available before the application continues.**

- Blocking also occurs with the `OutputStream` class where the application pauses until data is sent.
- **Can avoid blocking by calling the `available()` method before calling the `read()` method.**
- The **`available()` method** returns the number of bytes available to be read from the open connection.
- If bytes are available, call the `read()` method; otherwise avoid calling the `read()` method because it will create a block until bytes are available to be read by your program.
- Before reading downstream, call the **`mark()` method** to mark your current position in the stream.
- The **`mark()` method** requires an **`int` parameter** that specifies how many bytes you can read ahead, return to the marked position in the stream by calling the `reset()` method. A word of caution: marking may not be available in all implementations; therefore you should call the **`markSupported()` method**, which returns a boolean value indicating whether or not marking is supported.
- The **`close()` method** terminates the input stream, not the connection. Rarely will you need to call the `close()` method because the input stream is automatically closed when the instance of the `InputStream` is out of scope.

### **OutputStream Class :**

- The `OutputStream` class defines the basic interface for writing data to an open connection.
- **There are five methods in the `OutputStream` class, including three versions of the `write()` method.**
- These are
  - `write(intb)`, `write(byteb[])`, `write(byteb[],intoff,intlen)`,
  - `flush()`, and
  - `close()`.
- The **first version of the `write()` method** writes one byte to the output stream. This byte is passed as an `int` parameter to the `write()` method.
- The **second version of the `write()` method** writes multiple bytes to the output stream. Those bytes are contained in the byte array that is passed to the `write()` method.
- The **third version of the `write()` method** writes specified elements of a byte array to the output stream. **This version requires three parameters. The first parameter is the byte array. The second parameter is an `int` representing the offset of the first element that will be written to the output stream. The last parameter is an `int` representing the total number of bytes that are to be written.**
- The `OutputStream` class, like the `InputStream` class, uses blocking. In the case of the `OutputStream` class, blocking causes the application to wait until all pending bytes are written to the output stream.

- The **flush() method** causes any pending bytes to be output to the output stream.
- The **close() method** terminates the output stream, not the connection.



### HYPERTEXT TRANSFER PROTOCOL

- The communications protocol HTTP is supported by MIDP1.0 as required by specification. Support for other protocols is implementation dependent.
- HTTP requires that a client initiate a request for information from a remote computer.
- The remote computer is typically a server, although any computing device can respond to a client's request if the computing device supports HTTP.
- Communicating using HTTP is very similar in concept to sending and receiving an email. The client creates an email with a request for information.
- The email contains the address of the person who will supply the information and the return address of the client, which is used to respond to the email.
- The **Uniform Resource Locators (URLs)** of the client and remote computer are the addresses used in HTTP.
- The network operating system routes the request to the remote computer, where the request is reviewed and a response is returned to the client.
- Creating an HTTP Connection GCF optimizes mobile devices by providing a level of abstraction for network services enabling the device profile to select network protocols and network services to support.
- HTTP is used to connect to web pages.
- **An HTTP connection is made by calling one of the three versions of the open() method of the Connector class.**
- **All three versions require at least a connection string parameter that identifies the connection and throws an IOException error.**
- They also return a Connection. **Here's the syntax for the first version of the open() method: static Connection open (String connectString) throws IOException**

- The **second version of the open() method** requires a second parameter, which is an **int that identifies the mode in which the connection is opened.**
- **three connector modes: READ, WRITE, and READ\_WRITE.**
- The **READ mode** causes the connection to be used only for data input.
- The **WRITE mode** permits data to be sent but not received over the connection.
- The **READ\_WRITE mode** enables the connection to be used to send and receive data.
- **The syntax for the second version of the open() method:**

**static Connection open (String connectString, int mode) throws IOException**

- The **third version of the open() method** requires a third parameter consisting of a **boolean value that indicates whether or not the application can handle a timeout exception.**
- A **timeoutexception** is thrown whenever an attempt to create the connection fails.
- The timeout period and other aspects of the timeout exception are implementation dependent.
- **The method definition for the third version of the open() method:**

**static Connection open (String connectString, int mode, boolean timeouts) throws IOException**

- The connection string of each version of the open() method contains a unique identifier of the connection that conforms to the Uniform Resource Indicator (URI).
- The **identifier consists of three components: the scheme, the target, and parameters.** (Parameters are optional.)
- The **scheme** is the name of the network protocol used for the connection. Although you are guaranteed that an implementation supports the HTTP network protocol, some implementations also support socket, datagram, file, and port.

Mode	Description
READ	Open connection for read only
WRITE	Open connection for write only
READ_WRITE	Open connection for read and write

**Table 13-1.** CLDC Connector Modes

Network Protocol	Scheme	Target	Parameter
HTTP	http://	www.myweb.com	
Socket	socket://	www.mysocket:	1800
Datagram	datagram://	9000	
File	file://	myfile.txt	
Port	comm:	0;	baudrate=9600

**Table 13-2.** Values for the Connection String Parameter of the open() Method

- The **open() method** returns a Connection object that consists of the base interface for the connection. It is common to use the StreamConnection interface for an HTTPconnection.
- cast the Connection object returned by the open() method as a StreamConnection interface:

```
“StreamConnection connection = (StreamConnection)
Connector.open("http://www.myweb.com");”
```

### Reading Data from an HTTP Connection:

- Once opened an HTTP connection and obtained a stream using a connection interface, you are ready to read or write data using an input/output stream class.
- A common routine is for a J2ME application to request and then receive information from a web server, similar to the technique used by a browser to request and receive web pages.
- Aweb page is a text file that contains text and HTMLtags that describe how the text should be displayed on the screen.
- The browser interprets HTMLtags and displays the corresponding text accordingly.
- Interpretation of the contents is application dependent.
- The **initial step** in communicating with a remote computer is to open a connection and a connection interface.
- The **first line** of code opens an HTTPconnection to the web server www.jimkeogh.com and requests access to the index.htm file.
- The index.htm file is the home page of the web site. can replace the name of the web server and file with any web server and file that is available to you.
- Next, the openInputStream() method is called to create an instance of the InputStream.

```
StreamConnect connection = (StreamConnection)
Connector.open("http://www.jimkeogh.com/index.htm"); InputStream in =
connection.openInputStream();
```

- The connection and inputStream are like a highway between two computers. Once the highway is built, need to send cars along the highway. can do this by calling the read() method, assuming of course that you are reading data from the stream.
- The **read()** method returns an int. The value of the int corresponds to a character if your program is reading a text file.
- Probably one of the most common of these techniques is to assemble characters in a line of text.
- This code segment begins by creating a string buffer and an int.
- The read() method is called from within the conditional expression in the while loop.
- The return value of the read() method is an int, which is assigned to the ch int variable.
- The value of the ch variable is then compared to a -1 (negative one).
- A -1 is the value returned by the read() method when there isn't any data left to read from the stream. Next, the conditional expression in the if statement determines whether the value of the ch variable is the end-of-line character. If not, the append() method is called to append the character to the buffer.
- The buffer requires one parameter, which is a char. Since the read() method returns an int, need to convert the int to a char and pass the char to the append() method.
- If the value of the ch variable is the end-of-line character, the buffer contains the first line of the file and can be processed further within the program.
- can process the line of text any way you wish as necessary to fulfill the requirements of the application.
- After the line of text is processed, you should prepare the string buffer for the next series of characters by resetting the string buffer as shown in the last statement in this code example.

```
StringBuffer buffer = new StringBuffer(); int ch; while (( ch = in.read()) != -1) { if (ch != '\n') { buffer.append((char) ch);  
} else { System.out.println(buffer.toString()); buffer.delete(0, buffer.length()); }  
}
```

### Reading and Parsing a Web Page

- HTTP is the only protocol guaranteed to be supported by all implementations and is used to retrieve information from a web server.
- A web server can serve information in various kinds of formats, the most common being HTML.
- The **indexOf() method** returns an int that represents the character position within the line of the a in the alt= tag.
- This value is assigned to the int position in this MIDlet. Next the substring() method is called.

- The **substring() method** returns a subset of characters contained within the line. The subset is identified by the **two parameters passed to the substring() method**.
- The first parameter is an int representing the character position of the first character of the subset.
- The second parameter is an int representing the character position of the last character of the subset.
- The substring is then displayed in an alert dialog box. Notice that the content of the buffer is erased after each line is read from the input stream by assigning a reference to a new StringBuffer to the buffer variable.
- This assures there are no residual characters remaining in the buffer from the previous line. An IOException is thrown whenever the MIDlet is unable to connect to the web server. This error is displayed in an alert dialog box in this example.

### Retrieving Information from a Web Server:

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

public class HttpExample extends MIDlet implements CommandListener {
private Command exit, start;
private Display display;
private Form form;
private StringItem stars;
public HttpExample () {
display = Display.getDisplay(this);
exit = new Command("Exit", Command.EXIT, 1);
start = new Command("Start", Command.EXIT, 1);
form = new Form("Customer Ranking");
form.addCommand(exit);
form.addCommand(start);
form.setCommandListener(this);
```

```
}

public void startApp() throws MIDletStateChangeException {
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 == start) {
        StreamConnection connection = null; InputStream in = null; StringBuffer buffer = new
        StringBuffer();

        try { connection = (StreamConnection) Connector.open(
            "http://www.amazon.com/exec/obidos/tg/detail/-/007222472X");
            in = connection.openInputStream();

            int ch;
            while ((ch = in.read()) != -1) {
                if (ch != '\n') {
                    buffer.append((char)ch);
                }
            }
            else {
                String line = new String (buffer.toString());

                if(line.equals("out of 5 stars")) {
                    int position = line.indexOf("alt=");

                    Alert alert = new Alert( "Rating", line.substring(position + 5, position + 8), null, null);
                    alert.setTimeout(Alert.FOREVER);

                    alert.setType(AlertType.ERROR);
                }
            }
        }
    }
}
```



```
display.setCurrent(alert);
}
buffer = new StringBuffer();
}
}
}
catch (IOException error) {
    Alert alert = new Alert("Error", "Cannot connect", null, null);
    alert.setTimeout(Alert.FOREVER);
    alert.setType(AlertType.ERROR); display.setCurrent(alert);
}
}
}
}
```

### The File Protocol

- The file protocol is used to write information to a file from a MIDlet if the implementation supports the file protocol and if the device supports a file system.

#### Writing to a File:

```
import javax.microedition.midlet.*; import javax.microedition.lcdui.*; import java.io.*; import
javax.microedition.io.*; public class FileConnection extends MIDlet implements
CommandListener { private Command exit, start; private Display display; private Form form;
public FileConnection () { display = Display.getDisplay(this); exit = new Command("Exit",
Command.EXIT, 1); start = new Command("Start", Command.EXIT, 1); form = new
Form("Write To File"); form.addCommand(exit); form.addCommand(start);
form.setCommandListener(this); } public void startApp() throws MIDletStateChangeException
{ 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 == start) { try
{ OutputConnection connection = (OutputConnection)
Connector.open("file://c:/myfile.txt;append=true", Connector.WRITE ); OutputStream out =
connection.openOutputStream(); PrintStream output = new PrintStream( out ); output.println(
"This is a test." ); out.close();
```

```
connection.close(); Alert alert = new Alert("Completed", "Data Written", null, null);
alert.setTimeout(Alert.FOREVER); alert.setType(AlertType.ERROR); display.setCurrent(alert);

} catch( ConnectionNotFoundException error ) { Alert alert = new Alert( "Error", "Cannot
access file.", null, null); alert.setTimeout(Alert.FOREVER); alert.setType(AlertType.ERROR);
display.setCurrent(alert); } catch( IOException error ) { Alert alert = new Alert("Error",
error.toString(), null, null); alert.setTimeout(Alert.FOREVER);
alert.setType(AlertType.ERROR); display.setCurrent(alert); }

}

}

}
```

### SOCKET:

- A **socket** is a connection to a port on a remote computer that is used to exchange information using HTTP commands. HTTP commands enable your application to send an HTTP request for a stream of data to a remote computer.
- In return, the remote computer responds to your application by sending an HTTP response. Let's say your application wants to receive a stream of data from a particular file stored on a remote server. First application opens a socket connecting to a port on the server.
- Next, the application sends a GET command followed by the file name as part of an HTTP request. Your application then reads the stream containing the remote server's HTTP response.
- let's focus on opening a socket for two-way transmission with a remote server. A socket is created by calling the **Connector open() method** and passing it a string containing the socket schema.
- The socket schema consists of the socket identifier followed by the URL of the remote server. The URL must contain the port number of the remote server that is used to connect the remote server to the client.
- The following code segment illustrates a socket schema.

**socket://www.myserver.com:80**

- The `println()` method of the `PrintStream` is used to send strings containing HTTP commands to the remote server over the socket connection.
- The following code segment is a typical HTTP request. GET is the HTTP command used to request a file.
- **/my.html** is the name of the file being requested.
- **HTTP/0.9** is the version of the protocol being used for transmission, and
- **\r \n** are the carriage return and new line characters.

**GET /my.html HTTP/0.9\r\n**

### Writing to and Reading from a Socket Connection:

```
import javax.microedition.midlet.*; import javax.microedition.lcdui.*; import java.io.*; import
javax.microedition.io.*; public class socketconnection extends MIDlet implements
CommandListener { private Command exit, start; private Display display; private Form form;
public socketconnection () { display = Display.getDisplay(this); exit = new Command("Exit",
Command.EXIT, 1); start = new Command("Start", Command.EXIT, 1); form = new
Form("Read Write Socket"); form.addCommand(exit); form.addCommand(start);
form.setCommandListener(this); } public void startApp() throws MIDletStateChangeException
{ display.setCurrent(form); } public void pauseApp() { } public void destroyApp(boolean
unconditional) { } public void commandAction(Command command, Displayable displayable) {
if (command == exit) {
```

Complete Reference / J2ME: TCR / Keogh / 222710-9 / Chapter 13

```
destroyApp(false); notifyDestroyed();
```

```
} else if (command == start) { try { StreamConnection connection = (StreamConnection)
Connector.open("socket://www.myserver.com:80"); PrintStream output = new
PrintStream(connection.openOutputStream() ); output.println( "GET /my.html HTTP/0.9\r\n" );
output.flush(); InputStream in = connection.openInputStream(); int ch; while( ( ch = in.read() )
!= -1 ) { System.out.print( (char) ch ); } in.close(); output.close(); connection.close(); } catch(
ConnectionNotFoundException error ) { Alert alert = new Alert( "Error", "Cannot access
socket.", null, null); alert.setTimeout(Alert.FOREVER); alert.setType(AlertType.ERROR);
display.setCurrent(alert); } catch( IOException error ) { Alert alert = new Alert("Error",
error.toString(), null, null); alert.setTimeout(Alert.FOREVER);
alert.setType(AlertType.ERROR); display.setCurrent(alert); } } }
```

### Communication Management Using HTTP Commands :

- HTTP is used by two computers to communicate with each other over a socket connection.
- HTTP is considered a request/response protocol, meaning that a client is the requestor of information contained on a server and the server responds to the client's request.
- **An HTTP request, called a request entity, consists of three components. These are the request method, request header, and request body.**
- The **request method** describes the way in which information is to be sent to the client by the server. The client specifies the request method as part of the request.
- **There are three request methods .**
- **These are GET, POST, and HEAD.**
- **There are basically two ways in which data is sent—by including the data as part of the URL as a query string, which is called the GET method, or by sending data in a stream aside from the URL, called the POST method.**

- The **GET** method requires a URL and data separated by a question mark, which is also known as URLencoding.
- Data is grouped into one or more fields, each of which has a value associated with the field.
- A value is associated with a field by using an assignment operator (=), and each fieldvalue pair is separated by an ampersand (&).

Request	Description
GET	Send requested data as part of the URL
POST	Send requested data in a separate stream
HEAD	Send only meta-information about the specified resource

**Table 13-3.** *Types of HTTP Request Methods*

- Data sent in response to a request is assigned to an environment variable within the requestor's operating environment.
- The requestor then retrieves the data by reading the environment variable.
- The form of the data being returned using the GET request method.

<http://www.myclient.com/inventory?prod123=100&prod456=150>

- The **POST** method requires data to be sent in a separate stream, which provides two advantages over the GET method.
- Data sent using the GET method is visible whenever the URL is visible.
- Data attached to the end of the URL whenever we select the Submit button after filling out a form in a browser.
- The **POST method causes the data not to be visible as part of the URL.**
- **The other advantage of using the POST method is to be able to send an unlimited amount of data in response to a request.**
- The size of data sent using the GET method is limited by the maximum size of data that can be held by the environment variable.
- Data exceeding the size of the environment variable will probably be lost after reaching the requestor.
- The **POST** method is not a secure method to transmit data. It is simply less accessible than the GET method because data is sent on a different stream.
- The **HEAD method** requests metadata about the resource. Remember that metadata is information about data, such as the last time inventory data was updated.
- A MIDlet that wants to retrieve the latest inventory status of a product may do so only if the inventory recently changed. Therefore, the MIDlet uses the HEAD method and

follows up with a GET or POST method if the inventory has changed. Metadata is returned using the same technique as the GET request method. `HttpConnection` is used to send an `HTTPrequest` method.

```
HttpConnection connection = (HttpConnection)
```

```
Connector.open("http://www.myserver.com/inventory.dat");
```

- Once an `HttpConnection` is opened, you can use `HttpConnection` client request methods to interact with the remote server.
- **There are two ways to interact with the remote server—by setting or retrieving the request method, or by setting or retrieving the `HTTPHeader` information.**
- The request method is set by calling the `setRequestMethod()`, which requires a string containing **GET, POST, or HEAD** as a parameter.
- The `getRequestMethod()` is called to retrieve a string that contains the current request method.
- The `setRequestProperty()` method is used to write a value to a field in the `HTTP` header.
- **Two parameters are required by the `setRequestProperty()` method.**
- The **first parameter** is a string containing the field name, called a key, which is being set.
- The **other parameter** is a string containing the new value of the specified header field.
- A request header is information that describes the request using one or a combination of 40 header fields.
- Many header fields are used only rarely.
- **Here are commonly used header fields:**

- Accept
- Cache-Control
- Content-Type
- Expires
- IF-Modified-Since
- User-Agent

- Let's say that a client wants to receive a response using the plain text content type and using the GET request method.
- Here's the code segment that needs to be included in the MIDlet:

```
HttpConnection connection = (HttpConnection)
```

```
Connector.open("http://www.myserver.com/inventory.dat");
```

```
connection.setRequestMethod(HttpConnection.GET);
```

```
connection.setRequestProperty("Content-Type","//text/plain");
```

- The body of an HTTPrequest contains any information that is sent to the server as part of the request. This information is transferred to the server using either the GET or POST request method, as described in the request method portion of the request.
- The request method, request header, and the body of the request are transmitted over the connection to the server, where software running on the server reads each component of the request, processes the request based on the nature of the application, and returns a response with a response entity.
- **A response entity consists of three components: the status line, header response, and the body of the response.**
- The **status line** is a snapshot of the server's response to the client's request.
- **There are more than 35 response status codes for the HttpURLConnection.**
- **A three-digit number represents a status code.**
- The **first digit** represents the category of the status code and the remaining digits represent a specific status within a category.
- **There are two parts to the status line.**
- The **first part** is the status code and the other, the status message. You retrieve the status code by calling the **getResponseCode()**.
- The **getResponseCode()** returns an int whose value is the status code.
- The status message is retrieved by calling the **getResponseMessage()**.
- The **getResponseMessage() method** returns a string containing the status message

```
HttpConnection connection = (HttpConnection)
Connector.open("http://www.myserver.com/inventory.dat");
System.out.println(connection.getResponseCode());
System.out.println(connection.getResponseMessage());
```

- The response header contains header fields and related values similar to the request header .
- Can retrieve a header field or field value by calling methods of the HttpURLConnection class.
- **The most commonly used methods to retrieve the value of a header field are two versions of the getHeaderField() method.**
- The **first version** retrieves the value by referencing the index of the header field, which is passed to the **getHeaderField() as an int**.
- Each header field is assigned an index beginning with zero in the order in which the header field was entered into the header.
- Therefore, **getHeaderField(0)** retrieves the value of the first header field of the response header.

- The **secondversion** of the **getHeaderField()** retrieves the value of a header field by referencing the name of the header field, which is passed as a string to the **getHeaderField() method as a parameter.**
- can retrieve the name of a header field by calling the **getHeaderFieldInt()** method and passing it a string that contains the name of the header field.
- Likewise, you can retrieve the name of a header field by calling the **getHeaderFieldKey()** and passing it an int that represents the index of the header field.
- **Three useful headerfields** that are typically included in the response header are the **date, expiration, and lastmodified fields.**
- These are fairly self-explanatory:
- **date** contains the date when the header was created.
- **Expiration** is the date when information contained in the header is no longer valid.
- And the **lastmodified field** is the date when values in the header changed.
- **can retrieve these dates by calling the getDate() method, getExpiration() method, and getLastModified() method.**
- **All three return the date as along. Besides predefined response header fields, a response header can also include customized header fields.**
- **Customized header fields** are application specific, and their values are generated by the server-side process.
- The value assigned to a customized header field is retrieved by calling the **getHeaderField() method.**
- The **third component** of a response entity is the body, which contains the information that the client requested from the server.
- Http Connection does not contain methods for reading the body of a response entity because you read the body

Status Code Category	Description
100–199	Information
200–299	Success
300–399	Redirection
400–499	Client Error
500–599	Server Error

**Table 13-4.** *Response Status Code Categories*

### Using an HttpURLConnection to Communicate with a Server:

```
Import javax.microedition.lcdui.* import javax.microedition.midlet.*; import  
javax.microedition.io.*; import java.io.*; public class httpconnection extends MIDlet
```

```
implements CommandListener { private Command exit, start; private Display display; private
Form form; public httpconnection () { display = Display.getDisplay(this); exit = new
Command("Exit", Command.EXIT, 1); start = new Command("Start", Command.EXIT, 1); form
= new Form("Http Connection"); form.addCommand(exit); form.addCommand(start);
form.setCommandListener(this); } public void startApp() { display.setCurrent(form); } public
void pauseApp() { }
```

```
public void destroyApp(boolean unconditional) { destroyApp(false); notifyDestroyed(); } public
void commandAction(Command command, Displayable displayable) { if (command == exit) {
destroyApp(false); notifyDestroyed(); } else if (command == start) { HttpURLConnection connection
= null; InputStream inputstream = null; try { connection = (HttpURLConnection)
Connector.open("http://www.myserver.com/myinfo.txt"); //HTTP Request
connection.setRequestMethod(HttpURLConnection.GET); connection.setRequestProperty("Content-
Type", "text/plain"); connection.setRequestProperty("Connection", "close"); // HTTP Response
System.out.println(" Status Line Code: " + connection.getResponseCode()); System.out.println(
"Status Line Message: " + connection.getResponseMessage()); if
(connection.getResponseCode() == HttpURLConnection.HTTP_OK) { System.out.println(
connection.getHeaderField(0)+" "+connection.getHeaderFieldKey(0)); System.out.println(
"Header Field Date: " + connection.getHeaderField("date")); String str; inputstream =
connection.openInputStream(); int length = (int) connection.getLength(); if (length != -1) { byte
incomingData[] = new byte[length]; inputstream.read(incomingData); str = new
String(incomingData); } else { ByteArrayOutputStream bytestream = new
ByteArrayOutputStream();

int ch; while ((ch = inputstream.read()) != -1) { bytestream.write(ch); } str = new
String(bytestream.toByteArray()); bytestream.close();

} System.out.println(str);

}

} catch(IOException error) { System.out.println("Caught IOException: " + error.toString()); }
finally { if (inputstream != null) { try { inputstream.close(); } catch( Exception error) { /*log
error*/ } } if (connection != null) { try { connection.close(); } catch( Exception error) { /*log
error*/ } } }

}

}

}
```

### **Sending Data Along with an HTTP Request :**

- Typically, an HTTP request to a server is accompanied by information needed by the server to process the request.



- A case in point is when a server must authenticate a client using the client's user ID and password. The client sends the user ID and password along with the HTTP request.
- Data sent to a server must be in a pair value set, where the first element in the pair value is the field name and the second element is the value associated with the field.
- The field name and value must be separated by an equal sign. One of two techniques is used to send data to the server depending on whether the GET or POST request method is used for transmission.
- The GET request method requires that data be concatenated to the URL of the server.
- The POST request method requires that each pair value be written to the outputstream.

```
HttpConnection http = (HttpConnection) Connector.open("http://www.myserver.com/my.LoginServlet?UserID=jim&password=keogh;
```

- The flush() method is called after the last pair value set is written.

```
byte data[] = ("userID=jim"); outputStream.write(data);  
data = ("&password=Keogh"); outputStream.write(data);  
outputStream.flush();
```

### SESSION MANAGEMENT

- A key drawback of HTTP communication is the lack of persistence.
- HTTP is a stateless protocol, and there is no record of an HTTP transaction once a server responds to a client.
- The lack of state becomes a concern if multiple related HTTP transactions occur during the same session. This is the case in an order entry application.
- A session is created for one order. A separate HTTP transaction occurs each time an item is placed in the shopping cart and when the order is presented to the customer at the checkout counter.

```
String url = http.getHeaderField("SessionIDURL");
```

```
HttpConnection connection = (HttpConnection) Connector.open(url + "?" +  
"Prod=012345");
```

### Downloading an Image:

- From time to time, you will need to download an image from a server.
- The server-side application uses the session ID to associate the newly ordered item with other items ordered during the same session.
- **Cookies:** The other alternative to managing an HTTP session is to use a cookie. A cookie is a pair value in the HTTP response field and an HTTP request field that contains the session ID generated by the server-side application.

- Thesessionbeginswhenthe serverreceivestheclient'srequest. Theservergenerates thesessionID,assignsthesessionIDtothecookieHTTPresponsefield,andthensendsthe responsetotheclient.
- TheclientretrievesandtemporarilyretainsthesessionIDandthen processestheresponsefromtheserver.
- ThecookiepairvalueisthenwrittenasanHTTP requestfieldinthenextrequesttotheservermadebytheclient.
- Theserver-sideapplication retrievesthesessionID,andtheprocessisrepeateduntiltheclientterminatesthesession.

### **ContentConnection connection = (ContentConnection)**

**Connector.open("http://www.myserver.com/image/my.png");**

```
InputStream inputStream = (InputStream) Connector.openInputStream(connection); try {
ByteArrayOutputStream bytearray = new ByteArrayOutputStream(); int ch; while ((ch =
inputstream.read()) != -1) { bytearray.write(ch); } byte imagearray[] = bytearray.toByteArray();
// Create the image from the byte array Image image = Image.createImage(imagearray, 0,
imagearray.length); }
```

### **Transmitting a Background Process:**

- TheProcessclassdefinesthestart()method. Thestart()methodcreatesathreadand then calls the start() method to run the thread.
- The run() method calls the transmit() method.
- Thetransmit()methodcontainsallthecodetosendorreceiveatransmission.

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

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

```
public class BackgroundProcessing extends MIDlet implements CommandListener {
private Display display;
```

```
private Form form; private Command exit; private Command start; public
BackgroundProcessing() { display = Display.getDisplay(this); form = new Form("Background
Processing"); exit = new Command("Exit", Command.EXIT, 1); start = new Command("Start",
Command.SCREEN, 2); 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(false); notifyDestroyed(); } else if (command == start) { Process process = new
Process(this); process.start(); //Do foreground processing here } }
```

## Mobile Application Development Notes

---

```
} class Process implements Runnable { private BackgroundProcessing MIDlet; public
Process(BackgroundProcessing MIDlet)

{ this.MIDlet = MIDlet; } public void run() { try { transmit (); } catch (Exception error) {
System.err.println(error.toString()); } } public void start() { Thread thread = new Thread(this);
try { thread.start(); } catch (Exception error) { } } private void transmit() throws IOException {
//Place code here to receive or send transmission. }

}
```