CHAPTER 4:  
DESIGN AND CODING
Event Handling

Event processing is essential to most every MIDlet. It's hard to imagine an application where there is no user interaction whatsoever, even if this interaction is as simple as requesting to start or stop a MIDlet. Learning the classes that facilitate event processing is the goal of this chapter. We'll discuss only those events that apply to high-level user interface components. Low-level event handling (including key codes, game actions, and pointer events) will come in the next chapters.
To greatly oversimplify, event handling is nothing more than recognizing when an event occurs and taking an action based on that event. For example, recognizing that a help button has been pressed and displaying a help message. To break this down to the lowest level, there are three key steps to successfully managing an event.

1. The hardware (the physical device itself) must recognize that something has occurred: a button has been pressed, a button has been released, an adapter has been plugged in, an adapter has been removed. You get the picture.
2. The software on the device (the application manager) needs to be notified of the event.
3. This is the point where we come into the picture as a MIDlet developer. A message from the application manager will be sent to the MIDlet. This message will contain information about the event so we can make decisions as to how to proceed (e.g., the event may be a request to show a help message on the display).

Before a MIDlet can recognize a message from the application manager about an event, it must set up what we refer to as a "listener." There are two "listener" interfaces available in the MID Profile: CommandListener and ItemStateListener.

When you want to process events, you will need a class that implements one or both of these interfaces. As part of the class that implements the interface(s), you will write the method(s) commandAction() and/or itemStateChanged(). This is where you place the code to decipher what event occurred and how you would like to handle the event.

**Command Objects**

A Command is an object that holds information about an event. The simplest way to think of a Command is as a "button," something that you press or select. Figure 4.1 shows a
Command with the label "Exit." This Command is associated with the button directly below it on the device.

Figure 4.1. A Command mapped to a "generic" soft-button Screen shot

This idea, that a Command is a button, is a good starting point. However, because of limited screen space and the differences in availability of buttons on a device, it's not always that simple. We will explore this further as we delve into this chapter.

Processing events requires a little legwork up front. Here are the steps:
1. Create a Command object to hold information about an event.
2. Add the Command to a Form, Textbox, List or Canvas.
3. Add a "listener" to the above Form, Textbox, and so forth.
Upon detection of an event the "listener" will be called (sent a message). The end result is a call to the method commandAction(). Within this method you can determine what Command initiated the action and process the event accordingly.

Here are a few lines of code to allocate a Form add a Command to the Form and create a "listener" to detect events.

```java
private Form fmMain; // A Form
private Command cmExit; // A Command to exit the MIDlet
...
fmMain = new Form("Core J2ME"); // Form object
cmExit = new Command("Exit", Command.EXIT, 1); // Command object
...
fmMain.addCommand(cmExit); // Add Command to Form
fmMain.setCommandListener(this); // Listen for Form events
...
public void commandAction(Command c, Displayable s)
{
    if (c == cmExit)
    {
        destroyApp(true);
        notifyDestroyed();
    }
}
```
So, by considering the applications which we are going to design and further coding, first of all we design the Data Flow diagram that can bring the most concepts and ideas behind our four applications to design.

First we go through Omid - PhoneBook that is my first application to design and coding.

There is two different DFD for Phone Book, the first one is the Phone Book Process DFD and the second one is the Phone Book Search and different parts.

According to the Data Flow Diagram, when the phone book goes to run, the first process part is EDIT ORDER, which the user interface and selection of the user on menus is allocated there. So, the user can has an order to Add New contact to the list which use the second process part, so the user should enter a new first name, family name, and number, then the data will store in program owns database.

The third process part is the sorting part which user can sort the stored contact according to the First Name or Last Name.

The fourth process is browse process that will show all the stored contacts in the application.

There is one another part which will design in this application, and that is the Help menu part, that has brief information about the programmer and also a guideline to work with this application.
The most important part in this application is the search part which has come in the separate DFD, in search part by ordering the user to has a search, first should select according to the first name or last name or both, wants to search, so by entering there, now goes to the fourth process in the search DFD.

At this moment user should select that at Local (currently stored), wants to search or on the network wants to search.

If select the Local Search, the application search will has a search according to the input on the stored data to the database, and the result will send to the screen.
If not, the search technique should serve on the specific and particular server/IP/URL that has been stored previously on it, then the application will connect to that server/IP/URL and find according to the input and send the result to the screen.

Figure 4.3. Omid – Phone Book (Search Part) DFD
Labyrinth Design

The other application which is design here, is Omid – Labyrinth game. In this application I tried to design a simple game which could has interaction for the user also, even I looked for an application which can use whole the screen size and also could have change on the game size also.

The labyrinth game has to main process, in the first process, which user refer to it by selecting the Starting Game, will go through it, now the user has the options that can select to change its current labyrinth or can change the size of the game according to the game become easy or more tough, so the application should generate a new labyrinth according to the size that user has selected.

The other process which the user by enter to the application can select, is the Help process that gives the user a brief about the programmer, and also a complete help about how to play the game.

Figure 4.4. Omid – Labyrinth DFD
Hit Brick Design

The third application that is design is the Omid – Hit Brick game. In this game, I try to have more than a game functionality.

In the first process on the application, after the user enter to the application, can have an order that select by of the selective menu. So we go through to the second process which is the starting game, by selecting this menu, a new game will start and user should play the game, there is functionality that user can pause the game and again resume it.

The third process is the sound menu, in this menu user can select that want a game with the sound or not. User can turn on/off the game sounds.

The fourth process is the game score, the other functionality of this game that will make it as a unique one, is when the user finished a game, it will goes automatically to save its name and the score along with a photo capture facility. User can capture its own photo and then there is other facility that the user can send all these data to the specific server. So, there is a specific server that can store all these data in its database.

The fifth process is Invite process and part, in this process, user at first by entering the game can invite its friends to play this game, so by this invitation the user's friend will informed by receiving an SMS, so on they will start to play this game on their devices.

Finally when they send their data to the server, the server will sort them according to the high score order and the users can find out who get the highest score and is the winner.

There is one another part which will design in this application, and that is the Help menu part, that has brief information about the programmer and also a guideline to work with this application.
Figure 4.5. Omid – Hit Brick DFD
Drop AirForce Design

The fourth application that is design is the Omid – Drop AirForce game. In this game, I try to have more than a game functionality also.

The first process after the user enters to the game, is have an order that is choose a menu to work. The second process is the Starting Game. In this process user can start a new game by have the functionality of the using fire key during the game.

This game is designed for different steps and levels that the user can have more fun as well. This game also has the functionality that the user can pauses the game and again go through it and play it.

The third process in this application is Setting, in this part user can set that the sound of the game can be on/off and also the game vibration (when the user has dropping it will use) can be on/off.

The other process of this game is high score part, that if the user get a good score in the game, its score will save in the list, and user can see the top tens.

There is one another part which will design in this application, and that is the Help menu part, that has brief information about the programmer and also a guideline to work with this application.
Figure 4.6. Omid – Drop AirForce DFD
Now it is the time to go through different parts to start knowing of the coding for the design part.

A second type of event handling is done through an Item. An Item is any component that can be added to a Form. ChoiceGroup, DateField, Gauge and TextField are all subclasses of Item and each can process events. Items are accessible only as part of a Form, whereas Commands are available on Forms, as well as a Textbox, List, or Canvas.

Once you add an Item to a Form, as with Commands, you must add a listener. Once there is a change to an Item (e.g., a Gauge has been incremented or a DateField has been changed), the listener object will be notified (sent a message). Think of this message as a call to the method itemStateChanged(). This method can decipher what Item was modified and process the change as necessary.

Shown next is a small piece of code to create a Form, append a Date Field, and create a "listener" for event handling.

```java
private Form fmMain; // A Form
private DateField dfToday; // A DateField
...
fmMain = new Form("Core J2ME"); // Form object
dfToday = new DateField("Today: ", DateField.DATE); // DateField
...
fmMain.append(dfToday); // Add DateField to Form
fmMain.setItemStateListener(this); // Listen for events
...
public void itemStateChanged(Item item)
{
// If the datefield initiated this event...
if (item == dfToday)
{
}
```
When creating a new Command object to hold event information, there are three parameters: the label, type and priority. Using the following Command declaration, let's look at each parameter in detail.

```java
Command cmHelp = new Command("Help", Command.HELP, 1); // label type priority
```

Figures 4.7 shows a Command with the label "Exit." The Command is mapped to the softbutton shown directly below the label.

1. Label: This specifies the text that you would like to associate with the Command. The label may be shown directly on the screen or displayed inside a menu.
2. **Type:** If at all possible, we'd like to directly map Commands to relevant soft-buttons on a device. For example, if a device has a soft-button labeled "Help" it would be very intuitive for the user if there was a Command mapped to that soft-button to display a help message.

We specify our intent to map a Command to a specific softbutton using this parameter. For example, with the line of code shown above, we are specifying the intent of this Command to display help information.
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BACK</td>
<td>A request to move to the previous screen.</td>
</tr>
<tr>
<td>CANCEL</td>
<td>A request to cancel an operation. For example, when showing a screen to prompt for a web address, you may have both OK and CANCEL as options on the screen.</td>
</tr>
<tr>
<td>EXIT</td>
<td>A request to exit the MIDlet.</td>
</tr>
<tr>
<td>HELP</td>
<td>A request to display help information.</td>
</tr>
<tr>
<td>ITEM</td>
<td>A request to map the Command to an &quot;item&quot; on the screen. For example, when using a List component, you can mimic the functionality of a context-sensitive menu by mapping Commands to the various entries in the List.</td>
</tr>
<tr>
<td>OK</td>
<td>Specify positive acknowledgement from a user. For example, after downloading data, you may present a screen that says &quot;Download Complete&quot; with a Command of this type and a label &quot;OK.&quot;</td>
</tr>
<tr>
<td>SCREEN</td>
<td>For Commands in which it is unlikely there will be a specific key mapping available. For example, you might have Commands to initiate uploading and downloading of data. The labels &quot;Upload&quot; and &quot;Download&quot; will not have direct key mapping on a device.</td>
</tr>
<tr>
<td>STOP</td>
<td>A request to stop an operation. For example, if downloading data, this option may be available so a user can end the download without having to wait for the operation to complete.</td>
</tr>
</tbody>
</table>

Table 4.1. Shown the available types for this parameter

3. Priority: In relation to other Commands you define, this value represents where this Command falls in the line of priority. The higher number, has lower priority. These values may be helpful for the application manager when arranging items that appear in a menu or for ordering of soft-buttons on the display.

As with the type, the priority is only a request from you as a developer. The actual key mapping and priority assigned will be decided and handled by the device.
Table 4.2. Command Class: javax.microedition.lcdui.Command

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor</td>
<td></td>
</tr>
<tr>
<td>Command(String label, int commandType, int priority)</td>
<td>Create a new Command</td>
</tr>
<tr>
<td>Methods</td>
<td></td>
</tr>
<tr>
<td>int getCommandType()</td>
<td>Get type assigned to Command</td>
</tr>
<tr>
<td>String getLabel()</td>
<td>Get label assigned to Command</td>
</tr>
<tr>
<td>int getPriority()</td>
<td>Get priority assigned to Command</td>
</tr>
</tbody>
</table>

Table 4.3. CommandListener Interface: javax.microedition.lcdui.CommandListener

Example: Accessing Commands through Button or Menu

Previously it was mentioned that the easiest way to think of a Command is as a button on a device.

Example: Accessing Commands using in our 4 applications

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

public class AccessingCommands extends MIDlet implements CommandListener {

    private Display display; // Reference to Display
```
private Form fmMain; // A Form
private Command cmExit; // A Command to exit the MIDlet
public AccessingCommands()
{
    display = Display.getDisplay(this);
    cmExit = new Command("Exit", Command.EXIT, 1);
    fmMain = new Form("Core J2ME");
    fmMain.addCommand(cmExit);
    fmMain.setCommandListener(this);
}
// Called by application manager to start the MIDlet.
public void startApp()
{
    display.setCurrent(fmMain);
}
public void pauseApp()
{
}
public void destroyApp(boolean unconditional)
{
    // Check to see if our Exit command was selected
    public void commandAction(Command c, Displayable s)
    {
        if (c == cmExit)
        {
            destroyApp(false);
            notifyDestroyed();
        }
    }
Inside the constructor we create Command and Form objects, add the Command to the Form and set a "listener."

```java
cmExit = new Command("Exit", Command.EXIT, 1);
fmMain = new Form("Core J2ME");
fmMain.addCommand(cmExit);
fmMain.setCommandListener(this);
```

When the user interacts with the Form, such as pressing a button, the method `commandAction()` is called. We know from our previous discussion that this event started at the hardware level, worked its way through the application manager and finally arrived here.

```java
public void commandAction(Command c, Displayable s)
{
    if (c == cmExit)
    {
        destroyApp(false);
        notifyDestroyed();
    }
}
```

We check to see if the Command that was selected by the user (the value passed in as parameter "c") was the Exit Command. If so, we call destroyApp() to clean up any resources we acquired and follow this with a call to notifyDestroyed() to tell the application manager it is safe to shut down this MIDlet.
There should be no surprises in the output shown in Figure 4.9. The Form displays the title "Core J2ME" and the Command "Exit" is above a soft-button on the device. Pressing the button exits the MIDlet.

![Screen shot](image)

**Figure 4.9. Accessing the Exit Command with a soft-button Screen shot**

To complete that thought, the actual implementation of the Command is not defined by the API, only the characteristics and functionality. Put another way, how a Command is presented on the display, and whether it is mapped to a key or placed in a menu, is up to the device manufacturer.

**Example: Mapping Command to Buttons**

In the previous example, the handheld device placed our exit Command in a menu. Let's take a look at the Example, where a device maps a Command to a specific soft-button.
Example: MappingCommands using in our 4 applications

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

import javax.microedition.display.*;

public class MappingCommands extends MIDlet implements CommandListener {

private Display display; // Reference to Display

private Form fmMain; // The main Form

private TextBox tbHelp; // Textbox to display a help message

private Command cmExit; // Exit the MIDlet

private Command cmHelp; // Ask for Help

private Command cmBack; // Go "back" to main form

public MappingCommands() {

    display = Display.getDisplay(this);

    cmHelp = new Command("Help", Command.HELP, 1);

    cmBack = new Command("Back", Command.BACK, 1);

    cmExit = new Command("Exit", Command.EXIT, 1);

    // Create the Form, add Commands, listen for events

    fmMain = new Form("Core J2ME");

    fmMain.addCommand(cmExit);
```
public void startApp()
{
    display.setCurrent(fmMain);
}

public void pauseApp()
{
}

public void destroyApp(boolean unconditional)
{
    // Process events
    public void commandAction(Command c, Displayable s)
    {
        if (c == cmExit)
        {
            destroyApp(false);
            notifyDestroyed();
        }
        else if (c == cmHelp)
        {
            display.setCurrent(tblHelp);
        }
    }
    tblHelp = new TextBox("Help", "Help text here...", 25, 0);
    tblHelp.addCommand(cmBack);
    tblHelp.setCommandListener(this);
    // Called by application manager to start the MIDlet.
}
else if (c == cmBack)
    display.setCurrent(fmMain);
}

There are several additional components added to this MIDlet:

Textbox tbHelp; // Textbox to display a help message
Command cmHelp; // Ask for Help
Command cmBack; // Go "back" to main Form (from Help)

Let's see how the "traditional" cellular device displays the MIDlet. In Figure 4.10, the screen shot on the left is the main Form, with the Exit and Help Commands. When you press the Help button, the Textbox appears as shown by the screen shot on the right.

![Figure 4.10. Commands Exit and Back mapped to soft buttons Screen shot](image_url)
You can follow the logic of this MIDlet by looking at the commandAction() method:

```java
public void commandAction(Command c, Displayable s) {
    if (c == cmExit) {
        destroyApp(false);
        notifyDestroyed();
    } else if (c == cmHelp) {
        displaysetCurrent(tbHelp);
    } else if (c == cmBack) {
        displaysetCurrent(fmMain);
    }
}
```

**Example: Multiple Commands**

In our first example, we saw that the handheld device added the Exit Command to a menu. The reason was that there was no button that directly correlated to Exit. If we add more Commands without button mappings, the handheld will add these to the menu as well.

If you were to venture a guess, how do you think our cellular device will handle more Commands than can fit on the display? Let's look at Example and see.
Example: Multiple Commands using in our 4 applications

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

public class TooManyCommands extends MIDlet implements CommandListener
{

    private Display display; // Reference to Display

    private Form fmMain; // The main Form

    private TextBox tbAction; // Textbox upload/download

    private Command cmExit; // Exit the MIDlet

    private Command cmBack; // Go "back" to main form

    private Command cmUpload; // "upload" data

    private Command cmDownload; // "download" data

    // The constructor

    public TooManyCommands()
    {

        display = Display.getDisplay(this);

        cmExit = new Command("Exit", Command.EXIT, 1);

        cmBack = new Command("Back", Command.BACK, 1);

        cmUpload = new Command("Upload", Command.SCREEN, 2);

        cmDownload = new Command("Download", Command.SCREEN, 3);

        // Create the Form, add Commands, listen for events
        fmMain = new Form("Core J2ME");
    }
```
fmMain.addCommand(cmExit);
fmMain.addCommand(cmUpload);
fmMain.addCommand(cmDownload);
fmMain.setCommandLister(this);

// Create a Textbox, add Command, listen for events
Action = new TextBox("Process Data",
"Upload/download data ", 25, 0);
Action.addCommand(cmBack);
Action.setCommandLister(this);

} // Called by application manager to start the MIDlet
public void startApp()
{
    display.setCurrent(fmMain);
}

public void pauseApp()
{
}

public void destroyApp(boolean unconditional)
{

    // Process events
    public void commandAction(Command c, Displayable s)
    {
        if (c = cmExit)
        {
            destroyApp(false);
            notifyDestroyed();
        }
        else if (c = cmUpload | c == cmDownload)
            display.setCurrent(thAction);
        else if (c = cmBack)
            display.setCurrent(fmMain);
I have added two new Commands, cmUUpload and cmDLoad. Notice the priorities assigned.

```java
cmUUpload = new Command("Upload", Command.SCREEN, 2);
cmDLoad = new Command("Download", Command.SCREEN, 3);
```

By assigning "Upload" a higher priority than "Download" we are stating our preference regarding the relative importance of each Command. You'll see the end result of this request in Figure 4.11, where "Upload" appears before "Download."

![Figure 4.11. Commands mapped to both soft-buttons and a menu Screen shot](image)
Now that we've examined how events are managed within a MIDlet, we need to focus on how to create the user interface. Previously we introduced the Display and Displayable classes. If you recall, there is one Display object and any number of Displayable objects in a given MIDlet. The Screen is one of two Displayable objects defined as part of the core API, the other being Canvas.

This chapter will focus on the first of four components that are derived from the Screen class, the Form. It will also cover all the predefined components that may be added to a Form, including DateField, Gauge, StringItem, TextField, ChoiceGroup and ImageItem.

The Screen class itself is not something that you can see. Rather, Screen is a parent class for components that have an actual look and feel on the display. To further clarify, let's look at an illustration of the user interface hierarchy, starting with the Displayable class.

To further clarify the hierarchy shown in Figure 4.12, the class definitions follow:
Figure 4.12. Displayable class hierarchy

Display (public class Display)

Displayable (public abstract class Displayable)

Screen (public abstract class Screen extends Displayable)

TextBox (public class TextBox extends Screen)

List (public class List extends Screen implements Choice)

Alert (public class Alert extends Screen)

Form (public class Form extends Screen)

Item (public abstract class Item)
ChoiceGroup (public class ChoiceGroup extends Item)

implements Choice)

DateField (public class DateField extends Item)

TextField (public class TextField extends Item)

Gauge (public class Gauge extends Item)

ImageItem (public class ImageItem extends Item)

StringItem (public class StringItem extends Item)

Canvas (public abstract class Canvas extends Displayable)

Command (public class Command)

Ticker (public class Ticker)

Graphics (public class Graphics)

Choice (public interface Choice)

The MIDP includes two classes that are direct descendents of the Displayable class: Screen and Canvas. The Screen (and its subclasses) is for high-level user interface components; the Canvas is for custom graphics and event handling.

If we break this down further, you can see that Form, List, TextBox and Alert are all subclasses of the Screen class. Ticker (a scrolling ticker tape) is not a subclass; rather, it is a field (variable) defined inside the Screen class. Thus, a Ticker can be part of any Screen.

If you look carefully you'll notice that Item is not a subclass of Form. However, it is illustrated as such in Figure 4.12 to make the point that subclasses of Item are displayed on a Form.

Let's look at the methods available in the Screen class.
If you come from a desktop computing platform, the limitations of the display on a mobile device may take some getting used to. Overlapping windows, toolbars and cascading menus are nowhere to be found.

All is not lost, however. A Form gives you the option to show multiple components on the display.

The implementation will provide scrolling, as needed, to accommodate the components. Think of a Form as a container that can hold any number of components, where each is a subclass of Item (see Figure 4.12).

A Form has methods to append, insert, replace and delete components. When you append a component to a Form, the append method returns a numeric index indicating where the component was placed, with an index value starting at 0. Several other methods use this index as an input parameter; thus, you may find it helpful to keep a copy of this index in a variable. Other methods are available to determine the number of components on a Form and request the index (position) of a specific component.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String getName()</td>
<td>Get title associated with the Screen</td>
</tr>
<tr>
<td>void setTitle(String s)</td>
<td>Set title for the Screen</td>
</tr>
<tr>
<td>Ticker getTitle()</td>
<td>Get Ticker associated with the Screen</td>
</tr>
<tr>
<td>void setTicker(Ticker ticker)</td>
<td>Set Ticker for the Screen</td>
</tr>
</tbody>
</table>

Table 4.4. Screen Class: javax.microedition.lcdui.Screen
### Method

**Form (String title)**
Create a form

**Form (String title, Item[] items)**
Create a form and add Item(s) in the array

### Description

- **int append(Image img)**
  Append an Image

- **int append(Item item)**
  Append an Item

- **int append(String str)**
  Append a String

- **void delete(int itemNum)**
  Delete an Item

- **void insert(int itemNum, Item item)**
  Insert an Item prior to the Item specified

- **int get(int itemNum)**
  Get an Item

- **void setStateListener(ItemStateListener iListener)**
  Set (replace) an Item

- **int size()**
  Get the number of Items on a Form

### Table 4.5. Form Class: javax.microedition.lcdui.Form

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>An Item is a component that can be added to a Form. ChoiceGroup, DateField, Gauge, ImageItem, StringItem and TextField are all subclasses of Item. Working along with an Item is the class ItemStateListener. When you want to process events on an Item, you register a &quot;listener&quot; that will be sent a message when an event occurs. The recipient of this message is the method itemStateChanged(). Within this method, you can determine which Item changed and decide what action you would like to take. The following code appends a DateField to a Form and creates a listener for processing events.</td>
<td></td>
</tr>
<tr>
<td>private form frmMain; // A Form</td>
<td></td>
</tr>
<tr>
<td>private DateField dfToday; // A DateField</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
fmMain = new Form("Core J2ME"); // Form object

dfToday = new DateField("Today."); DateField.DATE; // DateField

fmMain.append(dfToday); // Add Item (DateField) to Form

fmMain.setItemStateListener(this); // Listen for Form events

public void itemStateChanged(Item item)
{
    if (item == dfToday)
    {
    }
}

Figure 4.13. Shows Labyrinth Installation Screen shot
Using the DateField component, you manipulate a Date object (as defined in java.util.Date) using the keys and/or soft buttons on a mobile device. The look and feel, as provided in Sun's MIDP implementation.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String getLabel()</td>
<td>Get the label assigned to the Item</td>
</tr>
<tr>
<td>void setLabel(String label)</td>
<td>Set label for the Item</td>
</tr>
</tbody>
</table>

Table 4.6. Item Class: javax.microedition.lcdui.Item

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DateField(String label, int mode)</td>
<td>Create DateField</td>
</tr>
<tr>
<td>DateField(String label, int mode, TimeZone timeZone)</td>
<td>Create DateField with specified TimeZone information</td>
</tr>
</tbody>
</table>

Table 4.7. DateField Class: javax.microedition.lcdui.DateField
Example: Creating an Alarm Clock

Let's create a simple alarm clock MIDlet (see Example). This application will allow the user to specify a date and time, and will sound an alarm and display a message when the appointed time has arrived.

To make this example a little more realistic, I've included two components that we have yet to introduce: the Timer and Alert. With a Timer you can schedule tasks to occur at some future time (in our case, a task to display a message). The message will be contained as part of an Alert component. An Alert is similar to a pop-up window or dialog box.

```java
public Snooze()
{

    display = Display.getDisplay(this);

    // The main form

    fmMain = new Form("When to sound the alarm");

    // Save today's date

    currentTime = new Date();

    // DateField with today's date as a default

    dfSnoozeTime = new DateField("", DateField.DATE_TIME);

    dfSnoozeTime.setDate(currentTime);

    // All the commands/buttons

    cmSnooze = new Command("Snooze", Command.SCREEN, 1);
    cmReset = new Command("Reset", Command.SCREEN, 1);
    cmExit = new Command("Exit", Command.EXIT, 1);

    // Add to form and listen for events

    datenonbox = fmMain.append(dfSnoozeTime);
    fmMain.addCommand(cmSnooze);
    fmMain.addCommand(cmReset);
```
```java
funMain.addEventListener(onExit);
funMain.setCommandListener(this);
funMain.setItemStateListener(this);
}
public void startApp ()
{
    display.setCurrent(funMain);
}
public void pauseApp()
{
}
public void destroyApp(boolean unconditional)
{
}
public void itemStateChanged(Item item)
{
    if (item == defSnoozeTime)
    {
        // If the user selected date and/or time that is
        // earlier than today, set a flag. We are using
        // getTime() method of Date class, which returns
        // milliseconds since January 1, 1970
        int snoozeTime = Date.getDefSnoozeTime().getTime();
        if (snoozeTime.getTime() < currentTime.getTime())
            dateOK = false;
        else
            dateOK = true;
    }
}
public void commandAction(Command c, Displayable s)
{
    if (c == CM_SNOOZE)
    {
```
if (dateOK == false) {
    Alert al = new Alert("Unable to set alarm", 
        "Please choose another date & time.", null, null);
    al.setTimeout(Alert.FOREVER);
    al.setType(AlertType.ERROR);
    display.setCurrent(al);
} else {
    // Create a new timer
    tmSnooze = new Timer();
    ttSnooze = new SnoozeTimer();
    // Amount of time to delay
    long amount = dfSnoozeTime.getDate().getTime() - 
        currentTime.getTime();
    tmSnooze.schedule(ttSnooze, amount);
    // Remove the commands
    fmMain.removeCommand(cmSnooze);
    fmMain.removeCommand(cmReset);
    // Remove the DateField
    fmMain.delete(dateIndex);
    // Change the form message
    fmMain.setTitle("Snoozing...");
} else if (c == cmReset) {
    // Reset to the current date/time
    dfSnoozeTime.setDate(currentTime = new Date());
}
If you've spent any amount of time on a computer, you've become accustomed to seeing progress meters in many shapes and forms. Familiar examples include a percentage indicator that is displayed when downloading a file or a progress meter shown when installing software. Should you need to provide a similar interface on a mobile device, the Gauge component may be the ticket.

A Gauge has two means of being updated. The first is referred to as interactive mode, where the user makes the changes. The second, for lack of a better term, is a non-interactive mode. It is up to you as the developer to change the values (see Figure 4.14).
4. Design and Coding

Figure 4.14. Interactive Gauge Screen shot

### Gauge API

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Constructor</strong></td>
<td></td>
</tr>
<tr>
<td>Gauge(String label, boolean interactive, int max\Value, int initial\Value)</td>
<td>Create a new gauge</td>
</tr>
<tr>
<td><strong>Methods</strong></td>
<td></td>
</tr>
<tr>
<td>int getValue()</td>
<td>Get current value of gauge</td>
</tr>
<tr>
<td>void setValue(int value)</td>
<td>Set new value for gauge</td>
</tr>
<tr>
<td>int getMaxValue()</td>
<td>Get maximum allowed gauge value</td>
</tr>
<tr>
<td>void setMaxValue(int max\Value)</td>
<td>Set maximum allowed gauge value</td>
</tr>
<tr>
<td>boolean isInteractive()</td>
<td>Is this an interactive gauge?</td>
</tr>
</tbody>
</table>

Table 4.8. Gauge Class: javax.microedition.lcdui.Gauge
A StringItem displays a static label and text message. A user cannot edit either the label or the text, and as a result, a StringItem does not recognize events.

As a developer, once you create a StringItem, you can get/set the text message with methods inside the StringItem class. If you would like to get/set the label, you can do so using methods inherited from the Item class.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor</td>
<td>StringItem(String label, String text)</td>
</tr>
<tr>
<td>Methods</td>
<td>String getText()</td>
</tr>
<tr>
<td></td>
<td>void setText(String text)</td>
</tr>
</tbody>
</table>

Table 4.9. StringItem Class: javax.microedition.lcdui.StringItem

Start with the picture in your mind of a single-line text-entry box. A common example would be a name or email address field that you might see on any standard online form. Now, add support for multiple lines of text and the option to filter the user input, such as only allowing numbers. At this point, you have a TextField component. If you are familiar with HTML form development, a TextField is similar to both a combination of a text input and text area, with a few twists.

When you create a TextField you can specify an input constraint. A constraint provides restrictions on the data that a user may enter. For example, you may have a TextField that prompts for an email address—the code behind the TextField can help by limiting the
characters it accepts to only those that are valid as part of an email address. There are four constraints to support the following specific types of input: email addresses, URLs, numeric values and phone numbers. There is an additional constraint that does no filtering at all, essentially passing all characters through to the TextField.

In addition to constraints, when you create a TextField you specify how many characters you anticipate you will need. As you might guess, there are no guarantees your requested size will be allocated; however, there is a method provided that will return the number of characters the TextField will support once created. Before calling a method that may extend the length of the TextField, save yourself some debugging time by checking the size before inserting data.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTRAINT_MASK</td>
<td>Use this mask when you need to determine the current value of the constraint. See the section entitled &quot;A Look Inside Constraint Values&quot; for more information.</td>
</tr>
<tr>
<td>ANY</td>
<td>Allow any character input.</td>
</tr>
<tr>
<td>EMAILADDR</td>
<td>Allow only characters that are valid within an email address.</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>Allow only numbers. This includes both positive and negative numbers. You do not have an option to request only positive or only negative values.</td>
</tr>
<tr>
<td>PASSWORD</td>
<td>Masks all character input to provide privacy when entering data. This constraint can be combined with other constraints to provide masking. See the section entitled &quot;Using the Password Modifier&quot; for more information.</td>
</tr>
<tr>
<td>PHONENUMBER</td>
<td>Allow only characters that are valid as part of a phone number. This may be device and/or local specific.</td>
</tr>
<tr>
<td>URL</td>
<td>Allow only characters that are valid within a URL.</td>
</tr>
</tbody>
</table>

Table 4.10. TextField Constraints: javax.microedition.lcdui.TextField

One last thought to keep in mind: The number of characters allocated for a TextField is not necessarily the same as the number of characters that will appear on the display. The implementation will add support for scrolling if the screen cannot display the text in its entirety.
### Table 4.11. TextField Class: javax.microedition.lcdui.TextField

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Constructor</strong></td>
<td>Create a new TextField</td>
</tr>
<tr>
<td>TextField(String label, String text, int maxSize, int constraints)</td>
<td></td>
</tr>
<tr>
<td><strong>Methods</strong></td>
<td></td>
</tr>
<tr>
<td>void delete(int offset, int length)</td>
<td>Delete characters at a specified offset</td>
</tr>
<tr>
<td>void insert(String src, int position)</td>
<td>Insert String at a specified offset</td>
</tr>
<tr>
<td>void insert(char[] data, int offset, int length, int position)</td>
<td>Insert specified characters from array into TextField at a specified offset</td>
</tr>
<tr>
<td>void setChars(char[] data, int offset, int length)</td>
<td>Set (replace) characters with data from array</td>
</tr>
<tr>
<td>void setString(String text)</td>
<td>Set (replace) TextField contents from String</td>
</tr>
<tr>
<td>int getChars(char[] data)</td>
<td>Get contents of TextField into an array</td>
</tr>
<tr>
<td>String getString()</td>
<td>Get contents of TextField into a String</td>
</tr>
<tr>
<td>int getConstraints()</td>
<td>Get constraints defined for TextField</td>
</tr>
<tr>
<td>void setConstraints(int constraints)</td>
<td>Set constraints for TextField</td>
</tr>
<tr>
<td>int getMaxSize()</td>
<td>Get max number of characters in TextField</td>
</tr>
<tr>
<td>int setMaxSize(int maxSize)</td>
<td>Set max number of characters in TextField</td>
</tr>
<tr>
<td>int getCursorPosition()</td>
<td>Get current caret (cursor) position</td>
</tr>
<tr>
<td>int size()</td>
<td>Number of characters currently in TextField</td>
</tr>
</tbody>
</table>

### Choice and ChoiceGroup

Before we can learn about the ChoiceGroup, we need to introduce the Choice interface. An interface is a class that defines a set of methods. It is up to the classes that "implement" the interface to provide the body of each method.

The Choice interface defines methods that all have to do with manipulating various types of predefined selections. There are two classes provided in the MIDP that implement the Choice interface:
A ChoiceGroup comes in two types: multiple and exclusive (think checkboxes and radio groups, respectively; Screen shots of both ChoiceGroups using Sun's reference implementation of MIDP are shown in Figure 4.15.

Figure 4.15. Multiple and exclusive ChoiceGroups Screen shot
Table 4.12. ChoiceGroup Class: javax.microedition.lcdui.ChoiceGroup

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor</td>
<td></td>
</tr>
<tr>
<td>ChoiceGroup(String label, int choiceType)</td>
<td>Create a ChoiceGroup with no elements</td>
</tr>
<tr>
<td>ChoiceGroup(String label, int choiceType, String[] stringElements, Image imageElements)</td>
<td>Create a ChoiceGroup and populate with data from the arrays</td>
</tr>
<tr>
<td>Methods</td>
<td></td>
</tr>
<tr>
<td>int append(String stringPart, Image imagePart)</td>
<td>Add element to end</td>
</tr>
<tr>
<td>void delete(int elementNum)</td>
<td>Delete element</td>
</tr>
<tr>
<td>void insert(int elementNum, String stringElement, Image imageElement)</td>
<td>Insert element</td>
</tr>
<tr>
<td>void set(int elementNum, String stringPart, Image imagePart)</td>
<td>Set (replace) element</td>
</tr>
<tr>
<td>String getString(int elementNum)</td>
<td>Get text (String) associated with element</td>
</tr>
<tr>
<td>Image getImage(int elementNum)</td>
<td>Get Image associated with element</td>
</tr>
<tr>
<td>int getSelectedIndex()</td>
<td>Get the index of the selected element</td>
</tr>
<tr>
<td>void setSelectedIndex(int elementNum, boolean selected)</td>
<td>MULTIPLE Choice Group-set element to specified boolean value</td>
</tr>
<tr>
<td>void setSelectedIndex(int elementNum, boolean selected)</td>
<td>EXCLUSIVE Choice Group-set element to true</td>
</tr>
<tr>
<td>int getSelectedFlags(boolean[] selectedArray)</td>
<td>IMPLICIT-invalid type for Choice Group</td>
</tr>
<tr>
<td>void setSelectedFlags(boolean[] selectedArray)</td>
<td>Store selection status in an array</td>
</tr>
<tr>
<td>boolean isSelected(int elementNum)</td>
<td>Set selection status from an array</td>
</tr>
<tr>
<td>int size()</td>
<td>Is the element currently selected?</td>
</tr>
<tr>
<td></td>
<td>Number of elements</td>
</tr>
</tbody>
</table>

Image and ImageItem

The ImageItem class allows you to specify how you would like an image displayed on a Form—for example, centered horizontally, to the left or to the right. However, your preference is not cast in stone. It is still up to the device implementation to decide where the
image will actually appear. When you consider the range of devices that may implement the MID Profile, this flexibility, from an implementation point of view, makes sense.

- Immutable: As the name implies, once you create this type of image, it cannot be changed. In most cases, you will create an immutable image from a resource, such as a file, or content stored in a data structure, such as a byte array. You can also convert a mutable image to immutable.

- Mutable: These image objects provide the ultimate in flexibility. When you create this type of image, you are doing nothing more than setting aside a chunk of memory that you will draw into, at some point, to create the image. An immutable image becomes visible when you specifically request it to be drawn to the display (e.g., using the paint() method as part of the Canvas class).

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static Image createImage(String name)</td>
<td>Create immutable image from resource</td>
</tr>
<tr>
<td>static Image createImage(Image source)</td>
<td>Create immutable image from existing Image</td>
</tr>
<tr>
<td>static Image createImage(byte[] imageData, int imageOffset, int imageLength)</td>
<td>Create immutable image from array data</td>
</tr>
<tr>
<td>static Image createImage(int width, int height)</td>
<td>Create mutable image</td>
</tr>
<tr>
<td>Graphics getGraphics()</td>
<td>Get reference to Graphics object for mutable image</td>
</tr>
<tr>
<td>int getHeight()</td>
<td>Get the height of image</td>
</tr>
<tr>
<td>int getWidth()</td>
<td>Get the width of image</td>
</tr>
<tr>
<td>boolean isMutable()</td>
<td>Determine if image is mutable</td>
</tr>
</tbody>
</table>

Table 4.13. Image Class: javax.microedition.lcdui.Image
### Table 4.14. ImageItem Layouts: javax.microedition.lcdui.Image

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAYOUT_DEFAULT</td>
<td>Use the default layout of the device implementation.</td>
</tr>
<tr>
<td>LAYOUT_LEFT</td>
<td>The image should appear on the left.</td>
</tr>
<tr>
<td>LAYOUT_RIGHT</td>
<td>The image should appear on the right.</td>
</tr>
<tr>
<td>LAYOUT_CENTER</td>
<td>Center the image horizontally.</td>
</tr>
<tr>
<td>LAYOUT_NEWLINE_BEFORE</td>
<td>Insert a newline before the image is drawn. This assures that previous text/images will appear above the image (not sharing the same horizontal space).</td>
</tr>
<tr>
<td>LAYOUT_NEWLINE_AFTER</td>
<td>Insert a newline after the image is drawn. Subsequent text/images will appear below the image.</td>
</tr>
</tbody>
</table>

### Table 4.15. ImageItem Class: javax.microedition.lcdui.ImageItem

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImageItem(String label, Image img, int layout, String altText)</td>
<td>Create an ImageItem</td>
</tr>
<tr>
<td>Methods</td>
<td></td>
</tr>
<tr>
<td>Image getImage()</td>
<td>Get Image associated with ImageItem</td>
</tr>
<tr>
<td>void setImage(Image img)</td>
<td>Set Image to be associated with ImageItem</td>
</tr>
<tr>
<td>int getLayout()</td>
<td>Get the current layout directives</td>
</tr>
<tr>
<td>void setLayout(int layout)</td>
<td>Set new layout directive</td>
</tr>
<tr>
<td>String getAltText()</td>
<td>Get alternate text to display if image cannot be shown on the device</td>
</tr>
<tr>
<td>void setAltText(String text)</td>
<td>Set alternate text to display if image cannot be shown on the device</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Directive</th>
<th>Decimal/Hex Value</th>
<th>Binary Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAYOUT_DEFAULT</td>
<td>0</td>
<td>00000000 00000000</td>
</tr>
<tr>
<td>LAYOUT_LEFT</td>
<td>1</td>
<td>00000000 00000001</td>
</tr>
<tr>
<td>LAYOUT_RIGHT</td>
<td>2</td>
<td>00000000 00000010</td>
</tr>
<tr>
<td>LAYOUT_CENTER</td>
<td>3</td>
<td>00000000 00000011</td>
</tr>
<tr>
<td>LAYOUT_NEWLINE_BEFORE</td>
<td>0x100</td>
<td>00000001 00000000</td>
</tr>
<tr>
<td>LAYOUT_NEWLINE_AFTER</td>
<td>0x200</td>
<td>00000010 00000000</td>
</tr>
</tbody>
</table>

Table 4.16. Layout Directives

It may not be obvious at first glance; however, it's important to understand that the first four directives were not intended to be combined (with each other). Although a bit contrived, here is an example combining LAYOUT_RIGHT with LAYOUT_CENTER:

```
ImageItem.LAYOUT_RIGHT 00000000 00000010
ImageItem.LAYOUT_CENTER 00000000 00000011

-----------------------------
logical OR 00000000 00000011
```

You've gained nothing by combining these directives. In fact, you are left with a directive equaling one of the original values, LAYOUT_CENTER.

On the other hand, the last two directives are specifically intended to be used along with other directives. For example:

```
ImageItem.LAYOUT_CENTER 00000000 00000011
ImageItem.LAYOUT_NEWLINE_BEFORE 00000001 00000000

-----------------------------
logical OR 00000001 00000011
```
Notice that the result is a unique value. That's a good thing. With this unique value the MIDP implementation can mask off the upper or lower eight bits and determine what directives were used to create the current value. For instance:

```
<table>
<thead>
<tr>
<th>Result from above</th>
<th>00000001</th>
<th>00000001</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mask off upper 8 bits</td>
<td>11111111</td>
<td>00000000</td>
</tr>
<tr>
<td>logical AND</td>
<td>00000001</td>
<td>00000001</td>
</tr>
</tbody>
</table>
```

The result tells us the original request was a directive combined with LAYOUT_NEWLINE_BEFORE.

Masking the other side:

```
<table>
<thead>
<tr>
<th>Result from above</th>
<th>00000001</th>
<th>00000001</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mask off lower 8 bits</td>
<td>00000000</td>
<td>11111111</td>
</tr>
<tr>
<td>logical AND</td>
<td>00000001</td>
<td>00000001</td>
</tr>
</tbody>
</table>
```

Referring back to Table 4.16, now, we know the entire directive that was originally requested—LAYOUT_CENTER combined with LAYOUT_NEWLINE_BEFORE.

This is important for two reasons. First, the MIDP implementation on a device needs to be able to correctly determine what directives you requested. Second, when you call the method `ImageItem.getLayout()`, you will be returned an integer value that is a combination of the directives currently assigned to the object. You will need to pull apart the integer value, by masking off bits, to determine what directives were applied when creating the ImageItem.

We can take this one step further:
Once again, we still have a unique value. The implementation can determine (by masking off bits) that we have requested a newline before drawing the image, as well as a newline after drawing the image.
All the user interface components we covered up to this point, with the exception of Form, were subclasses of Item. We are now about to introduce List, TextBox, Alert and Ticker. Each of these is on the same "level," if you will, as the Form. It's probably better to say that each extends the Screen class, as does the Form. The one exception is the Ticker class. Ticker cannot be a subclass of Screen. To be more precise, it is a variable defined in the Screen class. The actual declarations are as follows:

```java
public abstract class Screen extends Displayable {
    ...
    private Ticker ticker = null;
    ...
}
```

Thus, a Ticker can be associated with any (and actually with multiple) Screen object(s). Here are the class definitions for the hierarchy shown in Figure 4.16:
Figure 4.16. Displayable class hierarchy

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

Figure 4.17. Shows MenuItem and TextField and its Records of Labyrinth Game Screen shot

One significant difference between the components presented in this part and those covered thus far is how they are presented on the display. A Form can hold any number of Items. And along with that, a Form will handle any scrolling necessary to accommodate multiple components.

For example, a Form prompting a user for bank account information may contain several TextFields pertaining to an account number, email address and password. Each of these may be visible on the display at the same time. The Form will provide scrolling if necessary.
In contrast, a List, TextBox and Alert each operate independently. Once a TextBox is set as the active display, it is the only component visible. The same applies to an Alert and List. In essence, a Form is like a container for other visual components, whereas List, TextBox and Alert are standalone.

Bear in mind that whether we are referring to a Form, List or TextBox or Alert, each can accommodate Command objects as well. So when I state that TextBox is the only component on the display, this does not consider any Command objects that may be present.

**List**

A List contains a series of choices presented in one of three formats. Two of these are shown in Figure 4.18: multiple and exclusive.

![Figure 4.18. Multiple and Exclusive Lists Screen shot](image)
The third type is the implicit list (see Figure 4.19). Notice the simplicity of the list and the lack of radio buttons and checkboxes. I've also removed the "Save" command from the display. In a moment, you'll understand why such a command is no longer necessary.

![Implicit List Screen Shot](image)

**Figure 4.19. Implicit List Screen shot**

**Event Handling for List**

List and ChoiceGroup appear to be similar. The appearance is where the resemblance ends. Event handling for each is quite different. Let's quickly recap the event handling options for a ChoiceGroup:

1. Specify an ItemStateListener for the Form containing the ChoiceGroup. When a user changes a selection, the method itemStateChanged() is called.
2. Add a Command(s) to the Form containing the ChoiceGroup; for example, a "Save" or "Exit" Command. When a Command is issued by the user, the method commandAction() will be called.

Following are the event handling options for a List:

1. Exclusive and multiple lists: No event is triggered when a user changes a List element. To determine the status of the various elements, you will need a break in the action. Typically, this will be when a user selects a Command (as explained earlier for a ChoiceGroup).
   For example, you may have a multiple-selection list on the screen with various applications preferences, as shown in Figure 4.19. When a user chooses to "Save," you could write the preferences to persistent storage.

2. Implicit list: Upon selection of an element in the List, an event is immediately generated. As with exclusive and multiple, the method commandAction() will be called. Given what we know from previous run-ins with this method, it is only invoked if a CommandListener is registered. Following is a partial example to register a listener for an implicit List:

```java
List main = new List(Email Options, Choice.IMPLICIT);
main.setCommandListener(this);
```

The classic example of an implicit List is that of a menu. When presented with a series of options in a menu, once you make a selection the action is implicitly carried out. Put another way, once you choose an option from a menu, the menu goes away and you are whisked off to the code that carries out the action.

To put that description into a picture, visualize a pull-down menu inside a word processor. If you click on the Edit menu, you'll typically see options such as cut, copy, paste, search and replace. Once you've moved the mouse over your choice and clicked, the menu disappears and the action is carried out.
As a final clarification, unlike the ChoiceGroup, a List does not have the option of registering an ItemStateListener.

ChoiceGroup and List are so closely related that without a little nudge you may have a hard time differentiating between a List and a ChoiceGroup. Here are a few thoughts to help clarify:

**Similarities**

- Both implement the Choice interface. This interface defines a common set of methods: adding, deleting and getting elements, to name just a few.

- Both can show explicit choices (radio buttons) or multiple choices (check boxes).

**Appearance Difference**

- A ChoiceGroup appears as part of a Form. That is, there may be other components on the display. To be specific, there may be other objects of the Item class on the Form. Keep in mind that due to the limited display size, a user may need to scroll up and down to see all the components.

- A List occupies the display as a single entity. No other components will be visible at the same time. Once again, you may need to scroll up and down to see all the elements in the list, but the list is the sole component active on the display.

- One point to drill home: Whether we are referring to a Form (with a ChoiceGroup) or a List, both can accommodate Command objects. So when we say that a list is the only entity on the display, this does not take into consideration any commands that may be available. See Figure 4.20, which shows a list and two commands: "Exit" and "Add."
4. Design and Coding

Figure 4.20. Implicit List created with an array of Strings and an array of Images Screen shot

Implementation Differences

- ChoiceGroup can have the following types:
  MULTIPLE: Any number of elements can be selected (Checkbox)
  EXCLUSIVE: One element is selected at any given time (Radio Group)

- List can have the following types:
  MULTIPLE: Any number of elements can be selected (Checkbox)
  EXCLUSIVE: One element is selected at any given time (Radio Group)
  IMPLICIT: Selection of an element generates an event (Menu)

- Event processing for a ChoiceGroup can use either an ItemStateListener to detect changes, which will result in a call to the method itemStateChanged(), or you can have a Command object on a Form trigger a call to the method commandAction().
A List is not a subclass of Item: Therefore, there is no listener available to detect changes to individual elements. Events for an implicit list are generated as soon as a selection is made.

With a list you also have the option to use a Command object to initiate an event. Whether an implicit list or a command(s) trigger an event, the method commandAction() is called.

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>List(String title, int listType)</td>
<td>Create a new List with no elements</td>
</tr>
<tr>
<td>List(String title, int listType, String[] stringElements, Image[] imageElements)</td>
<td>Create a new List and populate with data from the arrays</td>
</tr>
<tr>
<td>int append(String stringPart, Image imagePart)</td>
<td>Add element to end</td>
</tr>
<tr>
<td>void delete(int elementNum)</td>
<td>Delete an element at specified index</td>
</tr>
<tr>
<td>void insert(int elementNum, String stringPart, Image imagePart)</td>
<td>Insert element at specified index</td>
</tr>
<tr>
<td>void set(int elementNum, String stringPart, Image imagePart)</td>
<td>Set (replace) element at specified index</td>
</tr>
<tr>
<td>String getString(int elementNum)</td>
<td>Get text of element at specified index</td>
</tr>
<tr>
<td>Image getImage(int elementNum)</td>
<td>Get Image of element at specified index</td>
</tr>
<tr>
<td>int getSelectedIndex()</td>
<td>Get index of selected element</td>
</tr>
<tr>
<td>void setSelectedIndex(int elementNum, boolean selected)</td>
<td>MULTIPLE LIST—set element to specified boolean value</td>
</tr>
<tr>
<td>int getSelectedFlags(boolean[] selectedArray, return)</td>
<td>EXCLUSIVE LIST—set element to true</td>
</tr>
<tr>
<td>void setSelectedFlags(boolean[] selectedArray)</td>
<td>IMPLICIT LIST—set element to true</td>
</tr>
<tr>
<td>boolean isSelected(int elementNum)</td>
<td>Store selection status in an array</td>
</tr>
<tr>
<td>int size()</td>
<td>Set selection status from an array</td>
</tr>
<tr>
<td></td>
<td>Determine if element is selected</td>
</tr>
<tr>
<td></td>
<td>Number of elements in List</td>
</tr>
</tbody>
</table>

Table 4.17. List Class: javax.microedition.lcdui.List
The List component implements the Choice interface. There are three pre-defined choice types, of which each is defined in Table 4.18.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXCLUSIVE</td>
<td>Only one selection available at any time</td>
</tr>
<tr>
<td>MULTIPLE</td>
<td>Zero or more selections available at any time</td>
</tr>
<tr>
<td>IMPLICIT</td>
<td>Selection of an element generates an event</td>
</tr>
</tbody>
</table>

Table 4.18. ChoiceTypes: javax.microedition.lcdui.Choice

Example: Implicit List

Let's look at Example, which creates an implicit list using an array of Image objects and an array of String objects.

```java
public implicitList() {
    display = Display.getDisplay(this);
    // Create the Commands
    cmExit = new Command("Exit", Command.EXIT, 1);
    cmAdd = new Command("Add", Command.SCREEN, 1);
    try {
        // Create array of image objects
        Image images[] = {Image.createImage("H.png"),
                          Image.createImage("rr.png"),
                          Image.createImage("new.png")};
        // Create array of corresponding string objects
        String options[] = {"Next", "Previous", "New");
```
// Create list using arrays, add commands, listen for events

isDocument = new List("Document Options",
List.IMPLICIT, options, images);
isDocument.addCommand(cmExit);
isDocument.addCommand(cmAdd);
isDocument.setCommandListener(this);
}
catch (java.io.IOException e)
{
System.err.println("Unable to locate or read .png file");
}

public void startApp()
{

display.setCurrent(isDocument);
}

public void pauseApp()
{

}

public void destroyApp(boolean unconditional)
{

}

public void commandAction(Command c, Displayable s)
{

// If an implicit list generated the event
if (c == List.SELECT_COMMAND)
{

switch (isDocument.getSelectedIndex())
{

case 0:
System.out.println("Next");
}
A TextBox is a multi-line text entry screen. When a user is entering text, you can filter the input to allow only certain characters (e.g., only numbers).

Does this component sound similar to another component we've seen previously? Just as a List and ChoiceGroup are closely related, so are a TextBox and TextField. So closely tied are the two that they share the same set of constraints for restricting the user input. The constraints are defined in the TextField class, and are shown in Table 4.19.

When you create a TextBox you specify how many characters you would like the TextBox to hold. The actual size returned may be different than what you requested. Therefore, a method is provided that will return the maximum number of characters: getMaxSize().
The difference between a TextBox and TextField come down to a few key points:

Similarities
- Both share the input constraints as specified in the TextField class.
- Both have a maximum capacity that may be different than what can be displayed on the device at any one time. Scrolling will be provided by the implementation as necessary.

Appearance Difference
- A TextField is a subclass of Item, and therefore appears as part of a Form. There may be other components on the Form at the same time.
- When on the display, a TextBox is the only component visible.

Implementation Differences
- When you want to process events on a TextField, you can use an ItemStateListener to detect changes that will result in a call to the method itemStateChanged() or you can have a Command object on the Form trigger a call to the method commandAction().
- TextBox events are handled exclusively through Command objects. Because a TextBox is not a subclass of Item, there is no ItemStateListener available.
## Constructors

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TextBox (String title, String text, int maxSize, int constraints)</code></td>
<td>Create a new TextField</td>
</tr>
</tbody>
</table>

## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void delete(int offset, int length)</code></td>
<td>Delete characters</td>
</tr>
<tr>
<td><code>void insert(String src, int position)</code></td>
<td>Insert characters from a String</td>
</tr>
<tr>
<td><code>void insert(char[] data, int offset, int length, int position)</code></td>
<td>Insert characters from an array into specified position</td>
</tr>
<tr>
<td><code>void setChars(char[] data, int offset, int length)</code></td>
<td>Replace TextBox contents with data from an array</td>
</tr>
<tr>
<td><code>int getChars(char[] data)</code></td>
<td>Place TextBox contents into an array</td>
</tr>
<tr>
<td><code>String getString()</code></td>
<td>Place TextBox contents into a String</td>
</tr>
<tr>
<td><code>void setString(String text)</code></td>
<td>Set (replace) TextBox contents from a String</td>
</tr>
<tr>
<td><code>int getConstraints()</code></td>
<td>Get the constraints defined</td>
</tr>
<tr>
<td><code>void setConstraints(int constraints)</code></td>
<td>Set the constraints</td>
</tr>
<tr>
<td><code>int getMaxSize()</code></td>
<td>Get total characters that can be stored</td>
</tr>
<tr>
<td><code>int setMaxSize(int maxSize)</code></td>
<td>Set total characters that can be stored</td>
</tr>
<tr>
<td><code>int getCaretPosition()</code></td>
<td>Get current input position</td>
</tr>
<tr>
<td><code>int size()</code></td>
<td>Number of characters currently in TextBox</td>
</tr>
</tbody>
</table>

### Table 4.19. TextBox Class: javax.microedition.lcdui.TextBox

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTRAINT_MASK</td>
<td>Use this mask when you need to determine the current value of the constraint. See the section &quot;A Look Inside Constraint Values&quot; in Chapter 7 for more information.</td>
</tr>
<tr>
<td>ANY</td>
<td>Allow any character input.</td>
</tr>
<tr>
<td>EMAIL_ADDR</td>
<td>Allow only characters that are valid within an email address.</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>Allow only numbers. This includes both positive and negative numbers. You do not have an option to request only positive or only negative values.</td>
</tr>
<tr>
<td>PASSWORD</td>
<td>Masks all character input to provide privacy when entering data. This constraint can be combined with other constraints to provide masking. See the section &quot;Using the Password Modifier&quot; in Chapter 7 for more information.</td>
</tr>
<tr>
<td>PHONENUMBER</td>
<td>Allow only characters that are valid as part of a phone number. This may be device and/or local specific.</td>
</tr>
<tr>
<td>URL</td>
<td>Allow only characters that are valid within a URL.</td>
</tr>
</tbody>
</table>

### Table 4.20. TextField Constraints: javax.microedition.lcdui.TextField
tbClip1 and tbClip2 will be instances of our new TextBoxCB class. Within each will be a reference to a Clipboard object (one object for each TextBoxCB). To make this example more intriguing, I've created just one character array to store text copied to a clipboard. This allows us to share the clipboard contents between any number of TextBoxCB objects. This is done by simply declaring the character array that holds the clipboard contents as a static variable. Thus, there is only one instance of _clipBoard, regardless of how many clipBoard objects we create.

**Alert and AlertType**

An Alert is a very simple dialog box that will support text and an Image (both are optional). The most common use of an Alert is to show a warning or error message.
Notice the three visual attributes you can control: title, image and text.

There are a few limitations to an Alert that you need to be aware of. First, Command objects (buttons) are not allowed on an Alert. User interaction is entirely managed by the implementation. In addition, no input can be solicited from a user. For example, you cannot add a TextField or TextBox to prompt for information.

There are two types of Alerts:
1. Modal: The Alert is on the display until a user dismisses it (see Table 4.21).
2. Timed: The Alert is on the display for a specific number of milliseconds.

Each device will have a default type for all alerts. You can determine the default by calling the method getDefaultTimeout(). A return value equivalent to Alert.FOREVER indicates modal; any other value indicates the time in milliseconds that an alert will be shown.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOREVER</td>
<td>Specifies the Alert is visible until acknowledged by the user (also known as modal).</td>
</tr>
</tbody>
</table>

Table 4.21. Alert Type: javax.microedition.lcdui.Alert

If you need to display an Alert type other than the default, call the method setTimeout(). The following creates a modal alert:

```java
Alert al;

al = new Alert("Error", "Error Message ", null, null);
al.setTimeout(Alert.FOREVER);
```

After you've created an Alert and set the attributes, there are two methods in the Display class to show the Alert.

```java
display.al.show();
```
1. `Display.setCurrent(Alert, Displayable)

    fmMain = new Form("Welcome");
    alertTest = new Alert("Alert", "Message Here ", null, null);
    alertTest.setTimeout(Alert.FOREVER);
    ...
    display.setCurrent(alertTest, fmMain);

This will display a modal alert. Once the alert has been acknowledged by the user, the alert will be removed from the display and the form, fmMain, will be displayed. The same logic applies if you change this to a timed alert. Once the alert has been displayed for the requested time, the alert will be replaced on the display by the form fmMain.

2. `Display.setCurrent(Alert)

    alertTest = new Alert("Alert", "Message Here ", null, null);
    alertTest.setTimeout(Alert.FOREVER);
    ...
    display.setCurrent(alertTest);

Presenting an Alert in this fashion assumes there is a Displayable object visible (a TextBox, List, Form or Canvas) when making this call. The reason is that, once the Alert has been dismissed (either acknowledged by the user or reached its requested display time), the previous Displayable will be shown.
In addition to the visual attributes, you can associate a sound with an Alert. This is done using the AlertType class.

There are five pre-defined sounds that are tied to various conditions: alarm, confirmation, error, info and warning (see Table 4.23). The idea is this: When you want to present an Alert and associated sound for general information, you would request the "info" sound. An Alert for an error condition would request the "error" sound, and so forth.

You can play a sound in one of two ways:

1. If the Alert has an associated AlertType, the sound will be played when the Alert is displayed. You can associate a sound by either specifying the AlertType as the last parameter to the constructor or using the method setType(). An example of each is shown here:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alert(String title)</td>
<td>Create a new Alert</td>
</tr>
<tr>
<td>Alert(String title, String alertText, Image alertImage, AlertType, alertType)</td>
<td>Create a new Alert with an Image and an associated sound (AlertType)</td>
</tr>
<tr>
<td>Image getImage()</td>
<td>Get Image associated with Alert</td>
</tr>
<tr>
<td>void setImage(Image img)</td>
<td>Associate an Image with Alert</td>
</tr>
<tr>
<td>String getString()</td>
<td>Get text associated with Alert</td>
</tr>
<tr>
<td>void setString(String str)</td>
<td>Set text for Alert</td>
</tr>
<tr>
<td>int getDefaultTimeout()</td>
<td>Get default time Alert is displayed</td>
</tr>
<tr>
<td>int getTimeout()</td>
<td>Get actual time Alert will be displayed</td>
</tr>
<tr>
<td>void setTimeout(int time)</td>
<td>Set amount of time to display Alert</td>
</tr>
<tr>
<td>AlertType getType()</td>
<td>Get the AlertType</td>
</tr>
<tr>
<td>void setType(AlertType type)</td>
<td>Set the AlertType</td>
</tr>
</tbody>
</table>

Table 4.22. Alert: javax.microedition.lcdui.Alert
Alert al;

al = new Alert("Alert", "Message ", null,
AlertType.WARNING);

Option #2:

Alert al;

al = new Alert("Alert", "Message ", null, null);

al.setType(AlertType.WARNING);

2. You can play a sound directly, without creating an Alert dialog box.

AlertType.INFO.playSound(myDisplay);

This may be helpful when you would like to signify something has taken place without changing what's on the display. For example, if a user requested to download data in the background, a simple confirmation sound when the download completed would be helpful. The AlertType is provided as an audible means to notify a user. Ideally, the sound would match the condition. For example, an error condition may trigger a louder and longer sound than that of a simple confirmation signifying a successful download. Bear in mind that a device may actually ignore requests to play a sound. Furthermore, if sounds are available, each is not guaranteed to be unique.
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALARM</td>
<td>Indicates arrival of a previous request to be notified</td>
</tr>
<tr>
<td>CONFIRMATION</td>
<td>Indicates completion of an event or action</td>
</tr>
<tr>
<td>ERROR</td>
<td>Indicates an error has occurred</td>
</tr>
<tr>
<td>INFO</td>
<td>Indicates general, non-critical information</td>
</tr>
<tr>
<td>WARNING</td>
<td>Indicates potential problem or situation</td>
</tr>
</tbody>
</table>

Table 4.23. AlertType Types: javax.microedition.lcdui.AlertType

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean playSound(Display display)</td>
<td>Play a sound on the device</td>
</tr>
</tbody>
</table>

Table 4.24. AlertType Method: javax.microedition.lcdui.AlertType

Example: SoundAlert using in Hit Break and Air Force Drop Game

```java
import javax.microedition.lcdui.*;
import javax.microedition.rp.cmm.*;

public class SoundAlert extends MIDlet implements ItemStateListener, CommandListener {
    private Display display; // Reference to display object
    private Form formMain; // Main form
    private Command cmExit; // Command to exit the MIDlet
    private ChoiceGroup cgSound; // Choice group

    public SoundAlert() {
        display = Display.getDisplay(this);
        // Create an exclusive (radio) choice group
        cgSound = new ChoiceGroup("Choose a sound", Choice.EXCLUSIVE);
        // Append options, with no associated images
        cgSound.append("Info", null);
    }

    public void handleCommand(Command c, Context ctx) {
        switch (c.getCommandType()) {
            case cmExit.getCommandType():
                System.exit(0);
                break;
            case cgSound.getCommandType():
                Sound sound = Sound.sound(cgSound.getString(c.getValue()));
                if (sound != null) {
                    sound.play();
                }
                break;
            default:
        }
    }

    public void itemStateChanged(Item item) {
        switch (item.getItemId()) {
            case cgSound.getCommandType():
                Sound sound = Sound.sound(cgSound.getString(cgSound.getSelectedCommand().getValue()));
                if (sound != null) {
                    sound.play();
                }
                break;
        }
    }
}
```

cgSound.append("Confirmation", null);
cgSound.append("Warning", null);
cgSound.append("Alarm", null);
cgSound.append("Error", null);
cmExit = new Command("Exit", Command.EXIT, i);

// Create Form, add components, listen for events
fmMain = new Form("/");
fmMain.append(cgSound);
fmMain.addCommand(cmExit);
fmMain.setCommandListener(this);
fmMain.setItemStateListener(this);
}

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

public void pauseApp()
{
}

public void destroyApp(boolean unconditional)
{
}

public void commandAction(Command c, Displayable s)
{
    if (c == cmExit)
    {
        destroyApp(false);
        notifyDestroyed();
    }
}

public void itemStateChanged(Item item)
{
switch (cgSound.getSelectedIndex())
{
    case 0:
        AlertType.INFO.playSound(display);
        break;
    case 1:
        AlertType.CONFIRMATION.playSound(display);
        break;
    case 2:
        AlertType.WARNING.playSound(display);
        break;
    case 3:
        AlertType.ALARM.playSound(display);
        break;
    case 4:
        AlertType.ERROR.playSound(display);
        break;
}

This MIDlet uses an exclusive ChoiceGroup (radio buttons) and registers an ItemStateListener for the Form that contains the ChoiceGroup. Bring to mind how this listener works. When a change is detected on an Item (ChoiceGroup is a subclass of Item), the method itemStateChanged() is called. Inside this method, we determine which entry in the choice was selected and play the appropriate sound:

public void itemStateChanged(Item item)
{
    switch (cgSound.getSelectedIndex())
    {
    case 0:
AlertType.INFO.playSound(display);
break;

---

Figure 4.22. High Definition of APIs in Labyrinth Game Screen shot

**Ticker**

Just as most C programmers wrote "Hello world" as their first program, Java has its own rite of passage—the scrolling text applet. The idea of sliding text across the display lives on. MIDP includes a scrolling text component known as the Ticker.
At this point we have covered all the components that extend the Screen class, making this an ideal time to introduce this class. A Ticker can be "attached" to any Screen, including an
Alert. In fact, the same Ticker can be shared among Screens, giving the illusion that the Ticker is attached to the device display, not the individual screens that make up the application.

Whether the text scrolls at the top or bottom and the speed at which scrolling occurs is decided by the implementation. However, you can be assured that the scrolling is continuous—as the trailing end of the text rolls off the screen, the beginning of the text will appear again.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ticker(String str)</td>
<td>Create a new Ticker with the specified ticker text</td>
</tr>
<tr>
<td>String getString()</td>
<td>Get the text associated with the Ticker</td>
</tr>
<tr>
<td>void setString(String str)</td>
<td>Set the text to associate with the Ticker</td>
</tr>
</tbody>
</table>

Table 4.25. Ticker Class: javax.microedition.lcdui.Ticker
Up to this point we've never given a second thought as to how a Form, TextBox or any other visual component appears on the display. We simply made a request to allocate and display a component, and left it at that. If the need never arose to move beyond "high-level" components we'd be set.

Then again, there is a whole new world on the other side. If we take the initiative to learn how to draw onto the display, the possibilities are nearly endless. Those interested in writing custom components and games will make extensive use of the classes presented in this chapter.

Two key classes make up the low-level API, Canvas and Graphics.

- The Canvas forms the backdrop. Not unlike an artists' canvas, it has a specific height and width and is drawn onto to create what the end user will see. Whatever is drawn becomes visible on the display. The canvas class also provides methods for low-level event handling.

- We draw onto a canvas with a Graphics object (often referred to as a graphics context). This class has methods for drawing lines, arcs, rectangles and text. This class also includes methods to specify color as well as font preferences. The exploration into creating a custom interface begins with the Canvas.

Let's start out at the outermost level and work our way in. From the big picture, a Canvas is a subclass of Displayable. Figure 4.23 revisits the user interface hierarchy.
Figure 4.23. Displayable class hierarchy

One last time, here are the class definitions for the hierarchy shown in Figure 4.23:

```
Display (public class Display)
Displayable (public abstract class Displayable)
Screen (public abstract class Screen extends Displayable)
TextBox (public class TextBox extends Screen)
List (public class List extends Screen and implements Choice)
Alert (public class Alert extends Screen)
Form (public class Form extends Screen)
Item (public abstract class Item)
ChoiceGroup (public class ChoiceGroup extends Item and implements Choice)
DateField (public class DateField extends Item)
```
There is only one Display object per MIDlet; however, there can be any number of Displayable objects. For example, a MIDlet can bounce between displaying a Form, followed by a TextBox, followed by a List, returning back to the Form. What is displayed and when is completely at our whim as the developer. We can now throw a Canvas into the mix, alternating between this Displayable and any other.

You won't directly request to display an instance of the Canvas class. Rather, you create a subclass of Canvas, and ask that this object be set as the current Displayable.

For example:

```java
class AnimationCanvas extends Canvas
{
    private Command cmExit; // Exit midlet
}```
cmExit = new Command("Exit", Command.EXIT, l);
addCommand(cmExit);
setCommandListener(this);

protected void paint(Graphics g)
{
    ...
}

AnimationCanvas canvas = new AnimationCanvas(this);
display.setCurrent(canvas);

The origin for drawing begins in the upper left corner of the display, location 0,0 (see Figure 4.24). X values increase heading to the right, y values increase heading down.
There are a few caveats worth mentioning. When working with a Graphics object we have the option to translate the origin as the Graphics object knows it. Second, when drawing text, an additional concept known as an anchor point is tossed in.

When drawing a line or shape (arc, rectangle, etc.), the thickness (known as the pen) is always 1 pixel wide. For example, if you request to start a line at 1,1, visualize a square pixel with a starting point at 1,1 and ending at 2,2 (always moving down and to the right). Even before moving the pen, a pixel has been filled! See Figure 4.25.
Canvas Width and Height

Table 4.26 shows the two methods to query a Canvas for its width and height. These values will always be the same for any one device. If a call to getWidth() returns 200, every Canvas created on that particular device will have a width of 200.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int getWidth()</td>
<td>Get the canvas width</td>
</tr>
<tr>
<td>int getHeight()</td>
<td>Get the canvas height</td>
</tr>
</tbody>
</table>

Table 4.26. Canvas Coordinate Methods: javax.microedition.lcdui.Canvas

When we are using a Canvas, we are responsible for drawing. Shortly thereafter I proceeded to show something similar to the following:

```java
AnimationCanvas canvas = new AnimationCanvas(this);
display.setCurrent(canvas);
```

This declares a Canvas and requests it to be set as the current displayable. This is the same path we took when requesting to display other components such as Form or TextBox.

Here's how it works. As explained earlier, Canvas is a subclass of Displayable. All displayable objects are made visible through:

```java
javax.microedition.lcdui.display.setCurrent(Displayable);
```

What differs between a Canvas and the other Displayable objects (Form, TextBox, List and Alert) is where the drawing of the component takes place and who is responsible for the drawing.

The Displayable class defines paint() as an abstract method. Following the rules defined by the Java language, both subclasses, Canvas and Screen, implement the paint() method. Here's
the primary difference: the paint() method inside Canvas is abstract, meaning there is no method body. It is up to a subclass to implement this method. Contrast this with the Screen, where the paint() method draws the title and Ticker (features common to all screens). Before leaving paint() the method paintContent() is called. Notice this method is declared as abstract, thus all subclasses (Form, TextBox, List and Alert) must implement this method.

```java
abstract public class Displayable {
    abstract void paint(Graphics g);
}

public abstract class Canvas extends Displayable {
    protected abstract void paint(Graphics g);
}

public abstract class Screen extends Displayable {
    {
        abstract void paintContent(Graphics g);
    }
}

public class Form extends Screen {
    {
        void paintContent(Graphics g) {
            ...
        }
    }
}

public class TextBox extends Screen {
    {
        void paintContent(Graphics g) {
            ...
        }
    }
}
Inside Form, TextBox, List and Alert paintContent() "knows" how to draw the component. For example, paintContent() inside List will draw a list on the device, the same method inside TextBox will draw a textbox, and so forth.

It boils down to this: Form, TextBox, List and Alert are drawn through the paint() method in the Screen class in concert with paintContent() inside each individual component. Canvas has declared paint() as abstract. Therefore, whenever we subclass Canvas, we are responsible for implementing this method.

Once again, here is a block of code that creates a new canvas:

```java
class AnimationCanvas extends Canvas
    implements CommandListener
{
    private Command cmExit; // Exit midlet
    ```
cmExit = new Command("Exit", Command.EXIT, I);
addCommand(cmExit);
setCommandLister(this);

protected void paint(Graphics g)
{
  // ... 
}

The paint() method is always passed a reference to a Graphics object, which is used to draw onto the Canvas. For example, to write text and draw a rectangle onto a Canvas, the paint method may look something like this:

protected void paint(Graphics g)
{
  g.drawString("Hello", 0, 0,
  Graphics.TOP | Graphics.LEFT);
  g.drawRect(5, 5, 10, 10);
  // ... 
}
paint() deserves top billing. This method serves as the workhorse for drawing to the display. You invoke the methods in the Graphics object (which is passed in as a parameter) to draw lines, arcs, rectangles and text.

repaint() is called to request the (entire) display to be updated. As they hold down the mouse button and move it around the display, we want to visually trace each step. We do this by calling repaint() after each movement, drawing a line segment from the last position to the current position.

The version of repaint() that accepts four parameters specifies a specific region on the display to repaint. We will visit this again when discussing clipping within the graphics section.

serviceRepaints() will force any pending paint requests to be immediately issued. Use this with caution, as this call will block until all paint requests have been serviced.

When the application manager is about to make a Canvas visible on the display, it will call the method showNotify(). When the Canvas has been removed from the display, the method hideNotify() is called.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paint(Graphics g)</td>
<td>Draw onto the canvas using the Graphics object specified</td>
</tr>
<tr>
<td>repaint()</td>
<td>Request the canvas to be painted</td>
</tr>
<tr>
<td>repaint(int x, int y, int width, int height)</td>
<td>Request a specified region of canvas to be painted</td>
</tr>
<tr>
<td>serviceRepaints()</td>
<td>Immediately process any pending paint requests</td>
</tr>
<tr>
<td>DoubleBuffered()</td>
<td>Does implementation provide double buffering?</td>
</tr>
</tbody>
</table>

Table 4.27. Canvas Paint Methods: javax.microedition.lcdui.Canvas

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>showNotify()</td>
<td>Application manager is about to show the canvas on the display</td>
</tr>
<tr>
<td>hideNotify()</td>
<td>Application manager has removed the canvas from the display</td>
</tr>
</tbody>
</table>

Table 4.28. Application Manager Communication: javax.microedition.lcdui.Canvas
Here is the code for each method taken directly from the Canvas class.

```java
protected void showNotify()
{
}

protected void hideNotify()
{
}
```

For example, inside `showNotify()` may be an ideal time to initialize variables, start a thread, and so forth. `hideNotify()` may be home to code to reset variables, stop a thread, and so forth. These methods provide an opportunity to take part in the transition as a Canvas moves from inactive to active, and back again, throughout the lifecycle of the MIDlet.

`showNotify()` will read an image from a file, set the preferred drawing coordinates and start a Thread. `hideNotify()` will reset a variable, effectively stopping the animation.

There are two general means of interaction between a user and the Canvas, Commands and low-level events. Commands. The low-level interface, consisting of key codes, game actions and pointer events, will be the prime focus of this section.

**Commands**

Canvas is inherited from Displayable. There are four methods available through this inheritance:

- `addCommand(Command)`
- `isShown()`
- `removeCommand(Command)`
- `setCommandListener(CommandListener)`
It can process Command objects just as we did with a Form, List and TextBox (Commands cannot be added to an Alert). The code for creating a Command, adding to a Canvas and registering a listener are identical to the same steps we've done numerous times before:

```java
class TestCanvas extends Canvas implements CommandListener {
    private Command cmExit; // Exit midlet
    ...
    cmExit = new Command("Exit", Command.EXIT, 1);
    addCommand(cmExit);
    setCommandListener(this);
    ...
    public void commandAction(Command c, Displayable d) {
        if (c.equals(cmExit))
            ...
    }
}
```

**Key Codes**

Key codes are numeric values that map directly to specific keys on a mobile device. The key codes that are guaranteed to be available on any MIDP are listed in Table 4.29. These codes correspond to the ITU-T keypad (0–9, *, #) the standard telephone keypad. Each key code has a static declaration similar to the following:

```java
public static final int KEY_NUM0 = 48;
public static final int KEY_NUM1 = 49;
...```
For example, when "1" is pressed on the keypad, the key code returned is 49 (KEY_NUM1).

The methods available for managing key codes are shown in Table 4.30.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void keyPressed(int keyCode)</td>
<td>Invoked when a key is pressed</td>
</tr>
<tr>
<td>void keyReleased(int keyCode)</td>
<td>Invoked when a key is released</td>
</tr>
<tr>
<td>void keyRepeated(int keyCode)</td>
<td>Invoked when a key is repeated</td>
</tr>
<tr>
<td>boolean hasRepeatEvents()</td>
<td>Does the implementation support repeated keys?</td>
</tr>
<tr>
<td>String getKeyKeyName(int keyCode)</td>
<td>The text string representing the key code</td>
</tr>
</tbody>
</table>

Table 4.30. Key Code Methods: javax.microedition.lcdui.Canvas

Game Actions

MIDP defines a set of constants, referred to as game actions, to facilitate event handling for game related events (see Table 4.31). Each game action is defined as a static integer:
public static final int UP = 1;
public static final int DOWN = 6;

You are assured that each game action will be assigned a key code by the implementation. For example, the game actions UP, RIGHT, LEFT, DOWN and FIRE may be mapped to the directional keys just below the display. Should a device not have such keys available, these same game actions may be mapped to the keypad using 2, 6, 4, 8 and 5, respectively.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Constant Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>UP</td>
<td>Move up</td>
<td>1</td>
</tr>
<tr>
<td>DOWN</td>
<td>Move down</td>
<td>6</td>
</tr>
<tr>
<td>LEFT</td>
<td>Move left</td>
<td>2</td>
</tr>
<tr>
<td>RIGHT</td>
<td>Move right</td>
<td>5</td>
</tr>
<tr>
<td>FIRE</td>
<td>Fire</td>
<td>8</td>
</tr>
<tr>
<td>GAME_A</td>
<td>Custom</td>
<td>9</td>
</tr>
<tr>
<td>GAME_B</td>
<td>Custom</td>
<td>10</td>
</tr>
<tr>
<td>GAME_C</td>
<td>Custom</td>
<td>11</td>
</tr>
<tr>
<td>GAME_D</td>
<td>Custom</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 4.31. Game Actions: javax.microedition.lcdui.Canvas

Detecting Game Actions

All low-level event handling on a Canvas is done through key codes (Commands, which are also available on a canvas, use the listener event model). Said another way, keyPressed(), keyReleased() and keyRepeated() are passed a key code when an event is triggered. Following are two options for working with game actions and their equivalent key codes:

Option #1

- During initialization, use getKeyCode() to request and store key codes for each game action.
- Inside keyPressed(), keyReleased() and keyRepeated(), branch based on the key Code Here is how this might look:
During initialization

```
keyFire = getKeyCode(FIRE);
keyRight = getKeyCode(RIGHT);
keyLeft = getKeyCode(LEFT);
```

At runtime

```
protected void keyPressed(int keyCode)
{
  if (keyCode == keyFire)
    shootWeapon();
  else if (keyCode == keyRight)
    moveRight();
  ```

Option #2
- Inside keyPressed(), keyReleased() and keyRepeated(), convert the incoming key code into the corresponding game action.
- Branch based on the game action
protected void keyPressed(int keyCode) {
    switch (getGameAction(keyCode)) {
    case Canvas.FIRE:
        shootWeaponO;
        break;
    case Canvas.RIGHT:
        moveRight();
        break;
    ...
    }

    Pointer Events
The MID Profile includes a set of methods to interact with pointer devices such as a mouse or touch screen.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean hasPointerEvents()</td>
<td>Does the platform support a pointer</td>
</tr>
<tr>
<td>boolean hasPointerMotionEvents()</td>
<td>Does the platform detect pointer motion (&quot;click/point and drag&quot;)</td>
</tr>
<tr>
<td>void pointerDragged(int x, int y)</td>
<td>Invoked when pointer dragged</td>
</tr>
<tr>
<td>void pointerPressed(int x, int y)</td>
<td>Invoked when pointer pressed</td>
</tr>
<tr>
<td>void pointerReleased(int x, int y)</td>
<td>Invoked when pointer released</td>
</tr>
</tbody>
</table>

Table 4.32. Pointer Methods: javax.microedition.lcdui.Canvas
The interaction with a pointer follows the same philosophy as with key codes. The methods pointerPressed(), pointerReleased() and pointerDragged() are invoked when a pointer is pressed, released or dragged. Each method is merely a placeholder and must be overridden if you choose to support pointer events. For example, you may choose to track the current x and y location as a pointer is moved about on the canvas.

```java
protected void pointerPressed(int x, int y)
{
    // Start location point where pointer pressed
    startx = x;
    starty = y;
}

protected void pointerDragged(int x, int y)
{
    // Current location of pointer
    currentx = x;
    currenty = y;
}

protected void pointerReleased (int x, int y)
{
    // End location point where pointer released
    endx = x;
    endy = y;
}
```
Figure 4.26. Shows the CanvasClass of Drop AirForce Game Screen shot

Graphics

A Graphic object is the instrument for drawing onto a Canvas. With over 30 methods, there is an abundance of operations from drawing of shapes and text to specifying fonts and colors. For consistency, code examples in this section will use the variable g to refer to an instance of a Graphics object. Whenever you see a reference to g, assume it has been properly allocated/acquired and always refers to a valid Graphics object, for instance:

```java
setColor(r, g, b);

drawLine(startx, starty, endx, endy);
```
There are two means to acquire a reference to this object. We witnessed the first when writing the paint() method inside the Canvas class. Here's a refresher:

```java
class AnimationCanvas extends Canvas

    implements CommandListener

{

    private Command cmExit; // Exit midlet

...

    cmExit = new Command("Exit", Command.EXIT, 1);
    addCommand(cmExit);
    setCommandListener(this);

...

    protected void paint(Graphics g)

    {
        ...

    }

}
```

The second means to obtain a Graphics object is through a mutable Image. As I discussed before the following should come for clarify the Graphic part as well.

**Immutable**: Once created, these images cannot be altered. Immutable images are most often created from a resource, such as a file. The most common use of an immutable image is to display an image as part of another component. In Figure 4.27 we created a ChoiceGroup...
with images that appear alongside each selection. A Form, List and Alert can also be home to immutable images.

**Figure 4.27. Immutable images inside a ChoiceGroup Screen shot**

**Mutable:** These images are created by requesting a block of memory, specifying the height and width of the desired image. The image is empty until drawn into. Here's how to allocate a mutable image:

```java
// Create mutable image and get graphics object for image
Image tmpImg = Image.createImage(80, 20);
Graphics g = tmpImg.getGraphics();
```

Look what we just uncovered. There's our second reference to a Graphics object, through a mutable image.
24 bits are allocated for color support; 8 bits to each of the colors: red, green, and blue. If a device does not support all possible colors in the range, it will map color requests to the closest possible match available. This includes mapping a color request to an appropriate shade of gray for non-color devices.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void setColor(int RGB)</td>
<td>Set color by combining each color component (Red, Green, Blue) into one integer value</td>
</tr>
<tr>
<td>void setColor(int red, int green, int blue)</td>
<td>Set color specifying each color component (Red, Green, Blue) separately</td>
</tr>
<tr>
<td>int getColor()</td>
<td>Get current color as one integer value</td>
</tr>
<tr>
<td>int getBlueComponent()</td>
<td>Get the blue component of the current color</td>
</tr>
<tr>
<td>int getGreenComponent()</td>
<td>Get the green component of the current color</td>
</tr>
<tr>
<td>int getRedComponent()</td>
<td>Get the red component of the current color</td>
</tr>
<tr>
<td>void setGrayScale(int value)</td>
<td>Set the grayscale</td>
</tr>
<tr>
<td>int getGrayScale()</td>
<td>Get current grayscale</td>
</tr>
</tbody>
</table>

Table 4.33. Color Support: javax.microedition.lcdui.Graphics

Setting Colors
When setting the color, we have two choices: combine the three color components into one integer value or specify each color as a separate integer. When combining colors, blue occupies the lower eight bits, followed by green, ending with red. For example, let's randomly select a value for each color:
When combining each separate color into one integer, we need to shift a few bits around. Specifically, red needs to move up 16 bits, green up 8 bits, with blue occupying the lower 8 bits.

```java
// Assume 'g' is a valid Graphics object

int red = 0,
green = 126,
blue = 255;

g.setColor((red << 16) | (green << 8) | blue);
```

The end result is an integer value with the binary representation:

```
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1
```

Red | Green | Blue

The second option is to specify the colors separately:

```java
g.setColor(red, green, blue);
```

**Getting Colors**

To get the current color configuration the options are the reverse of setting the colors: the current color selection can be returned as one integer value or a separate integer for each color.

For separate color values use `getRedComponent()`, `getGreenComponent()` and `getBlueComponent()`. When requesting the color is returned as one integer value, you can get the value assigned to each color (red, green and blue) by masking off the appropriate bits.
int colors, red, green, blue;

colors = g.getColor();

// Return the highest 8 bits
red = colors & 0xFF0000

// Return middle eight bits
green = colors & 0xFF00;

// Return lowest 8 bits
blue = colors & 0xFF

Using Grayscale

setGrayScale(int) allows selection of a shade of gray in the range of 0 to 255. As with a color device, if the value is outside the supported range, an appropriate match will be selected.

When drawing lines, arcs and rectangles we can choose between a solid or dashed stroke style. If you don’t specify a preference, the default is solid. Table 4.34 shows a line drawn in each stroke style.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOLID</td>
<td>Draw solid lines</td>
</tr>
<tr>
<td>DOTTED</td>
<td>Draw dotted lines</td>
</tr>
</tbody>
</table>

Table 4.34. Stroke Styles: javax.microedition.lcdui.Graphics
Each line has a starting point, \((x_1, y_1)\) and ending \((x_2, y_2)\) point (see Table 4.35). Regardless of the stroke style, the thickness of a line is always one pixel wide.

The starting location of the line and the current location of the mouse were specified as the parameters to `drawLine()`.

```java
// Draw with black pen
g. setColor(0, 0, 0);
g.drawLine(startx, starty, currentx, currenty);
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void drawLine(int x1, int y1, int x2, int y2)</code></td>
<td>Draw line specifying starting and ending points</td>
</tr>
</tbody>
</table>

Table 4.35. Line Drawing: `javax.microedition.lcdui.Graphics`

There are two methods to draw arcs: one to create an "outline" of an arc and the second to fill an arc (see Table 4.36).

```java
void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle) {
    Draw an arc inside a bounding box specified by x, y and width, height.
}

void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle) {
    Fill an arc inside a bounding box specified by x, y and width, height.
}
```

Table 4.36. Arc Methods: `javax.microedition.lcdui.Graphics`

Drawing an arc begins by specifying the bounding box (i.e., the outside dimensions of an "imaginary" box that will contain the arc). The startAngle is the location to start the arc,
where 0 is found at 3 o'clock. Positive values go counter-clockwise. Therefore, if you choose a startAngle of 90 the arc would begin at 12 o'clock.

The arcAngle is how many degrees to rotate from the startAngle. A startAngle of 0 and arcAngle of 180 would begin at 3 o'clock and rotate counter-clockwise to 9 o'clock. A startAngle of 90 with an arcAngle of 180 would begin at 12 o'clock and rotate counterclockwise to 6 o'clock.

Rectangles can have either square or rounded corners and can be drawn as an outline or filled (see Table 4.37).

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void drawRect(int x, int y, int width, int height)</td>
<td>Draw a rectangle</td>
</tr>
<tr>
<td>void drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)</td>
<td>Draw a rounded rectangle</td>
</tr>
<tr>
<td>void fillRect(int x, int y, int width, int height)</td>
<td>Fill a rectangle</td>
</tr>
<tr>
<td>void fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)</td>
<td>Fill a rounded rectangle</td>
</tr>
</tbody>
</table>

Table 4.37. Rectangle Methods: javax.microedition.lcdui.Graphics

The parameters for a non-rounded rectangle should be unmistakable. Specify the starting x and y location as well as the width and height. When creating a rectangle with rounded corners we also specify the horizontal diameter (arcWidth) and the vertical diameter (arcHeight) of the arc drawn at each corner.

Anytime we draw text we must take into consideration the font style currently assigned to the Graphics context. Before going any further, this is an opportune time to learn about fonts.
Fonts

Font support within MIDP has been slimmed down significantly compared with J2SE. The most glaring omission is the lack of a Font Metrics class. Metrics define characteristics of a Font that deal with measurements such as the height of a character or the size of the gap between one character and the next. MIDP does provide various metrics information. A new Font is requested through the static method Font.getFont().

```java
Font font = Font.getFont(Font.FACE_SYSTEM,
Font.STYLE_PLAIN,
Font.SIZE_MEDIUM);
```

The idea behind using a static method is in an effort to conserve resources, specifically, limiting garbage collection. When requesting a Font keep in mind that the implementation may not be able to accommodate your request. If this is the case, the nearest match will be returned.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static Font getFont(int face, int style, int size)</td>
<td>Request a new Font</td>
</tr>
<tr>
<td>static Font getDefaultFont()</td>
<td>Request the system Font</td>
</tr>
</tbody>
</table>

Table 4.38. Font: javax.microedition.lcdui.Font

There are three attributes associated with a Font: the face, style and size. One attribute deserves special attention—the style. Unlike the face and size, you can combine style attributes using a logical OR (|) operator. Here is a request for the system (default) face, which has both a bold and italic style and is medium in size.

```java
Font font = Font.getFont(Font.FACE_SYSTEM,
Font.STYLE_BOLD | Font.STYLE_ITALIC,
Font.SIZE_MEDIUM);
```
There are seven methods for querying a font's attributes, and each is listed in Table 4.39. One method worth mentioning is `getStyle()`, which returns an integer that may be any combination of the style attributes.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Constant Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FACE_SYSTEM</td>
<td>System characters</td>
<td>0</td>
</tr>
<tr>
<td>FACE_MONOSPACED</td>
<td>Monospace characters</td>
<td>32</td>
</tr>
<tr>
<td>FACE_PROPORTIONAL</td>
<td>Proportional characters</td>
<td>64</td>
</tr>
<tr>
<td>STYLE_PLAIN</td>
<td>Plain characters</td>
<td>1</td>
</tr>
<tr>
<td>STYLE_BOLD</td>
<td>Bold characters</td>
<td>2</td>
</tr>
<tr>
<td>STYLEITALIC</td>
<td>Italicized characters</td>
<td>4</td>
</tr>
<tr>
<td>STYLE_UNDERLINED</td>
<td>Underlined characters</td>
<td>8</td>
</tr>
<tr>
<td>SIZE_SMALL</td>
<td>Small characters</td>
<td>16</td>
</tr>
<tr>
<td>SIZE_MEDIUM</td>
<td>Medium characters</td>
<td>0</td>
</tr>
<tr>
<td>SIZE_LARGE</td>
<td>Large characters</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 4.39. Font Attributes: `javax.microedition.lcdui.Font`

```
Font font = Font.getFont(
    FontFACESYSTEM,
    FontSTYLEBOLD | FontSTYLEITALIC | FontSTYLEUNDERLINED,
    FontSIZE_MEDIUM);

int style = font.getStyle();
```

The variable `style` will have the value 7. Here's why.
If you would like to test for various style attributes you may write something as follows:

```java
if (style == (Font.STYLE_BOLD | Font.STYLE_ITALIC))
```

We get the same end result using the methods isPlain(), isBold(), isItalic() and isUnderlined(), such as:

```java
if (font.isBold() && font.isItalic())
```

With that in mind, let's assume that we prompt a user for their preferred font attributes, with the intention of saving these attributes in the Record Store. We could store the user preferences as three integer values:

```java
int face = font.getFace();
int style = font.getStyle();
int size = font.getSize();
```

`getStyle()` neatly packages all the preferred style attributes (bold, italic and underlined) into one variable, `style`. Without this method we would need to query for each style attribute (`isBold()`, `isItalic()` and `isUnderlined()`) and either store a reference to each one or go through the hoops to create one variable that will represent a combination of each.

To complete this example, let's assume that each time the MIDlet is started we read user preferences from the Record Store and update the system accordingly. To set the Font to the
user-preferred setting, we could read the attributes previously written and with one call set the preferred Font:

```java
int face, style, size;

// A custom method to read font attributes from rms
getFontAttributes();

// Request the user preferred font
Font font = Font.getFont(face, style, size);
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int getFace()</td>
<td>Get the current face</td>
</tr>
<tr>
<td>int getStyle()</td>
<td>Get the combination of style attributes as one integer (logically or'ed)</td>
</tr>
<tr>
<td>int getSize()</td>
<td>Get the current size</td>
</tr>
<tr>
<td>boolean isPlain()</td>
<td>Is the plain (style) attribute set</td>
</tr>
<tr>
<td>boolean isBold()</td>
<td>Is the bold (style) attribute set</td>
</tr>
<tr>
<td>boolean isItalic()</td>
<td>Is the italic (style) attribute set</td>
</tr>
<tr>
<td>boolean isUnderlined()</td>
<td>Is the underlined (style) attribute set</td>
</tr>
</tbody>
</table>

Table 4.40. Attributes Methods: javax.microedition.lcdui.Font

Metrics describe information regarding various measurements of a Font. Before going any further we need to understand the terminology use when working with metrics (see Figure 4.28).
The ascent is the distance from the baseline to the top of the majority of characters. The descent is the distance from the baseline down to the bottom of the majority of characters (some may extend below this). Leading is the gap between the descent of one line and the ascent of the next. The font height is defined as the ascent + leading + descent, which makes up the distance between baselines.

The advance is the total "distance" occupied by a character or string of characters. The advance is more than just the width of the individual characters; it also takes into consideration the gap between characters. The advance is useful when centering or otherwise horizontally aligning text.

Unlike the FontMetrics class in J2SE, MIDP does not include methods to obtain values for all metrics. However, the most common metrics for a font are available. getHeight() returns the font height. Included is the leading, which provides for a gap between successive lines of text. getBaselinePosition() returns the ascent, the distance from the baseline to the top of the tallest character. The remaining methods determine the advance (total length occupied) for a single character, array of characters, String and a sub-string of a String.

**Anchor Point**

There is just one last topic to cover before drawing text: the anchor point. When drawing text, in addition to providing the String to display, we also specify an x and y location. Given that the coordinate system begins in the upper left corner of the display, it would be a reasonable assumption that this x and y value would be an offset from that location.
That would work fine; however, to provide for additional flexibility and make it easier to align text, the concept of an anchor point was introduced. Anchor points are defined in pairs, just like x and y coordinates. The x (horizontal) values are LEFT, HCENTER and RIGHT. The y (vertical) values are TOP, BASELINE and BOTTOM. Picture these pairs as points around an imaginary box, known as the bounding box, of the text you would like to draw (see Figure 4.29).

Figure 4.29. Anchor points around the bounding box of a text String

<table>
<thead>
<tr>
<th>Anchor Point</th>
<th>Description</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEFT</td>
<td>Left of text</td>
<td>Horizontal</td>
</tr>
<tr>
<td>HCENTER</td>
<td>Center of text</td>
<td>Horizontal</td>
</tr>
<tr>
<td>RIGHT</td>
<td>Right of text</td>
<td>Horizontal</td>
</tr>
<tr>
<td>TOP</td>
<td>Top of text</td>
<td>Vertical</td>
</tr>
<tr>
<td>BASELINE</td>
<td>Baseline of text</td>
<td>Vertical</td>
</tr>
<tr>
<td>BOTTOM</td>
<td>Bottom of text</td>
<td>Vertical</td>
</tr>
</tbody>
</table>

Table 4.41. Text Anchor Points: javax.microedition.lcdui.Graphics
When you specify an anchor point what you are referring to is which location on the bounding box will be located at the x and y coordinate. We are putting the cart before the horse; however, to show an example using anchor points, let's look at the method drawString().

```java
        g.drawString("core j2me", 0, 0, Graphics.TOP | Graphics.LEFT);
        g.drawString("core j2me", 0, 0, Graphics.TOP | Graphics.CENTER);
```

The first parameter is obvious—this is the text to draw. The next two values are the x and y coordinates. The last parameter is the anchor point. Anchor points are always specified in pairs, a vertical and horizontal value, combined using the logical OR operator (|).

The first call requests that the TOP/LEFT corner of the bounding box be located at 0,0. The second requests that the TOP/HCENTER of the bounding box be located at 0,0 (see Figures 4.30).

To get a perspective of the coordinate system and the layout of the text, assume the light gray box represents the display.

```
0,0
```

- `core j2me` text on the display

Figure 4.30. Anchor point TOP/LEFT on bounding box at 0,0
This requests that the BASELINE / HCENTER of the bounding box be positioned at the center of the display (which is located by dividing both the screen width and height by 2).
The end result, for all intents and purposes, is that the text is centered.
The idea behind anchor points is to make our life a little easier when laying out text. Be patient, it takes a little trial and error to get the feel for anchor points.

Drawing Text
At this point, with an understanding of fonts and anchor points, drawing text is nothing more than deciding which method to call. We can opt to draw a single character, an array (or subset of an array) of characters, a String or subset of a String.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void drawChar(char character, int x, int y, int anchor)</td>
<td>Draw one character</td>
</tr>
<tr>
<td>void drawChars(char[] data, int offset, int length, int x, int y, int anchor)</td>
<td>Draw an array (or subset) of characters</td>
</tr>
<tr>
<td>void drawString(String str, int x, int y, int anchor)</td>
<td>Draw a String</td>
</tr>
<tr>
<td>void drawSubstring(String str, int offset, int len, int x, int y, int anchor)</td>
<td>Draw a sub-string of a String</td>
</tr>
<tr>
<td>Font getFont()</td>
<td>Get the current Font</td>
</tr>
<tr>
<td>void setFont(Font font)</td>
<td>Set the current Font</td>
</tr>
</tbody>
</table>

Table 4.42. Drawing Text Methods: javax.microedition.lcdui.Graphics

Regardless of the method you choose, the implementation will align the text based on the x and y coordinates and the anchor point.

Drawing of images is not unlike drawing a line or text. There is an appropriately named method drawImage(). However, there's a catch. Look over the parameters for the method and see if anything comes to mind:
The concern has to do with the first parameter, which is a reference to an Image object. This implies, and correctly so, that we can’t call drawImage() until we create an Image. Going one step further, if we create a mutable Image, which is nothing more than a block of memory, it has no content until drawn onto. It comes down to this: we need to lay the groundwork before calling drawImage().

Here are the basic steps:

**Immutable Image**

1. Allocate the image

   ```java
   Image im = Image.createImage("imageTest.png");
   ```

2. Display the image

   ```java
   protected void paint(Graphics g)
   {
     ...
     g.drawImage(im, 10, 10, Graphics.LEFT | Graphics.TOP);
     ...
   }
   ```
Mutable Image

1. Allocate the image

   ```java
   Image im = Image.createImage(80, 20);
   ```

2. Create the image content (using arcs, rectangles, lines and text)

   ```java
   // Get Graphics object to draw onto image
   Graphics graphics = im.getGraphics();
   // Draw a filled rectangle
   graphics.fillRoundRect(0, 0, 50, 50, 20, 20);
   ```

3. Display the image

   ```java
   protected void paint(Graphics g)
   {
   
   g.drawImage(im, 10, 10, Graphics.LEFT | Graphics.TOP);
   
   }
   ```
Example: Immutable Image using in our 3 Games

```java
public class ImmutableImage extends MIDlet {

    private Display display; // The display
    private ImageCanvas canvas; // Canvas

    public ImmutableImage() {
        display = Display.getDisplay(this);
        canvas = new ImageCanvas(this);
    }

    protected void startApp() {
        display.setCurrent(canvas);
    }

    protected void pauseApp() {
    }

    protected void destroyApp(boolean unconditional) {
    }

    public void exitMIDlet() {
        destroyApp(true);
    }
```
notifyDestroyed();

Figure 4.31. Shows the Alert, Vibration and Graphics in Drop AirForce Game Screen shot

<table>
<thead>
<tr>
<th>Anchor Point</th>
<th>Description</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEFT</td>
<td>Left of image</td>
<td>Horizontal</td>
</tr>
<tr>
<td>HCENTER</td>
<td>Center of image</td>
<td>Horizontal</td>
</tr>
<tr>
<td>RIGHT</td>
<td>Right of image</td>
<td>Horizontal</td>
</tr>
<tr>
<td>TOP</td>
<td>Top of image</td>
<td>Vertical</td>
</tr>
<tr>
<td>VCENTER</td>
<td>Center of image</td>
<td>Vertical</td>
</tr>
<tr>
<td>BOTTOM</td>
<td>Bottom of image</td>
<td>Vertical</td>
</tr>
</tbody>
</table>

Table 4.44. Image Anchor Points: javax.microedition.lcdui.Graphics
One final note: when drawing images the last parameter specifies the anchor point.

```java
    g.drawImage(im, 10, 10, Graphics.TOP | Graphics.LEFT);
```

As with text, the anchor point is the location on the bounding box that is positioned at the specified x and y coordinates. For example, the aforementioned anchor point requests that the top left corner of the bounding box be located at the coordinates 10, 10. Table 4.45 lists the horizontal and vertical anchor points available for images.

![Image anchor points](image)

**Figure 4.32. Image anchor points**

When defining the coordinate system we placed 0,0 at the upper left corner of the display.

Through a concept referred to as translation, a Graphics object can reference its origin (0,0) at a different location on the display.
For each call we've made to paint(), the entire display has been redrawn. A clipping region can be used to limit what is painted on the display. One important benefit is a reduction in the time to refresh the display.

Before we define a clipping region of our own, let's display an image along with information about the default clipping region.

```
Image im = Image.createImage("bridge.png");
...

protected void paint(Graphics g)
{

  System.out.println("x: " + g.getClipX());
  System.out.println("y: " + g.getClipY());
  System.out.println("width: " + g.getClipWidth());
  System.out.println("height: " + g.getClipHeight());

g.drawImage(im, 0, 0, Graphics.LEFT | Graphics.TOP);
}
```
For each call we've made to paint(), the entire display has been redrawn. A clipping region can be used to limit what is painted on the display. One important benefit is a reduction in the time to refresh the display.

Before we define a clipping region of our own, let's display an image along with information about the default clipping region.

```java
image im = Image.createImage("/bridge.png");

protected void paint(Graphics g)
{
    System.out.println("x: "+ g.getClipX());
    System.out.println("y: "+ g.getClipY());
    System.out.println("width: "+ g.getClipWidth());
    System.out.println("height: "+ g.getClipHeight());
    g.drawImage(im, 0, 0, Graphics.LEFT | Graphics.TOP);
}
```
That's interesting—the there is already a clipping region defined, even before we call setClip(). The default is a clipping rectangle that consists of the entire display—x and y starting at 0,0 and width and height matching the values returned by Canvas.getWidth() and Canvas.getHeight().

Let's modify the clipping rectangle and see how this affects the output on the display.

```java
protected void paint(Graphics g)
{
    g.setClip(25, 25, 45, 45);
    g.drawImage(im, 0, 0, Graphics.TOP | Graphics.LEFT);
}
```
Figure 4.33. Shows the Methods and Anchor Application support in Drop AirForce Game

Screen shot

Example: Clip

```java
public class Clip extends MIDlet {

private Display display; // The display
private ClipCanvas canvas; // Canvas

public Clip() {
    display = Display.getDisplay(this);
    canvas = new ClipCanvas(this);
}

protected void startApp() {
    display.setCurrent(canvas);
}
```
protected void pauseApp()
{
}
protected void destroyApp(boolean unconditional)
{
}
public void exitMIDlet()
{
destroyApp(true);
notifyDestroyed();
}

public ClipCanvas(Clip midlet)
{
this.midlet = midlet;
// Create exit command & listen for events
cmExit = new Command("Exit", Command.EXIT, 1);
addCommand(cmExit);
setCommandListener(this);
// Get random values for starting point
random = new java.util.Random();
// Make sure the entire clipping region is
// visible on the display
clipx = Math.min((getWidth() - clipw),
(random.nextInt() >>> 1) % getWidth());
clipy = Math.min((getHeight() - cliph),
(random.nextInt() >>> 1) % getHeight());
try
{
// Create immutable image
im = Image.createImage("/house.png");
}
catch (java.io.IOException e)
protected void paint(Graphics g)
{
if (im != null)
{
// Clear only the previous clipping region
    g.setColor(255, 255, 255);
g.fillRect(old_cx, old_cy, old_w, old_h);
// Set the new clipping region
    g.setClip(clipx, clipy, clipw, cliph);
// Draw image
    g.drawImage(im, 0, 0, Graphics.LEFT | Graphics.TOP);
}