Designing Java Applications for Series 60

Version 1.0
August 2002
# Table of Contents

1. **Introduction** ........................................................................................................... 6  
   1.1 Purpose and Scope .................................................................................................. 6  
2. **Overview** ............................................................................................................... 7  
   2.1 MIDlet Overview ................................................................................................... 7  
   2.2 Basic Design Considerations ............................................................................... 8  
      2.2.1 MIDlet limitations ......................................................................................... 8  
      2.2.2 MIDlet UIs .................................................................................................... 8  
3. **Application Life Cycle** .......................................................................................... 9  
   3.1 MIDlet Construction ............................................................................................. 9  
   3.2 StartApp() ............................................................................................................. 9  
   3.3 PauseApp() ........................................................................................................... 10  
   3.4 DestroyApp(boolean unconditional) ...................................................................... 10  
4. **UI Classes** .............................................................................................................. 11  
   4.1 Overview .............................................................................................................. 11  
   4.2 Navigation ............................................................................................................. 11  
5. **High-Level UI API** ............................................................................................... 12  
   5.1 Command Handling ............................................................................................. 12  
   5.2 Command Types .................................................................................................. 12  
      5.2.1 Exit command ............................................................................................... 12  
      5.2.2 Screen command ......................................................................................... 13  
      5.2.3 Item command ............................................................................................. 13  
      5.2.4 Help command ............................................................................................ 13  
      5.2.5 Back command ............................................................................................ 13  
      5.2.6 OK/Cancel command .................................................................................. 13  
   5.3 Designing Commands: ......................................................................................... 13  
   5.4 The Screen Class .................................................................................................. 14  
      5.4.1 TextBox ........................................................................................................ 14  
      5.4.2 List ................................................................................................................ 15  
      5.4.3 Alert .............................................................................................................. 17  
      5.4.4 Form .............................................................................................................. 17  
      5.4.4.1 ChoiceGroup ............................................................................................ 18  
      5.4.4.2 DateField .................................................................................................. 18  
      5.4.4.3 Gauge ........................................................................................................ 18  
      5.4.4.4 StringItem ................................................................................................ 19  
      5.4.4.5 TextField ............................................................................................. 19
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.5</td>
<td>Practical User Interface Design Steps</td>
<td>19</td>
</tr>
<tr>
<td>6.</td>
<td>Low-Level UI API</td>
<td>21</td>
</tr>
<tr>
<td>6.1</td>
<td>Low-Level Event Handling with Canvas</td>
<td>21</td>
</tr>
<tr>
<td>6.1.1</td>
<td>ShowNotify()</td>
<td>21</td>
</tr>
<tr>
<td>6.1.2</td>
<td>HideNotify()</td>
<td>21</td>
</tr>
<tr>
<td>6.1.3</td>
<td>KeyPressed(), keyRepeated(), and keyReleased()</td>
<td>21</td>
</tr>
<tr>
<td>6.2</td>
<td>Canvas Graphics</td>
<td>22</td>
</tr>
<tr>
<td>6.3</td>
<td>Graphics</td>
<td>22</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Displaying text</td>
<td>22</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Image class</td>
<td>22</td>
</tr>
<tr>
<td>7.</td>
<td>Timers and Threads</td>
<td>23</td>
</tr>
<tr>
<td>7.1</td>
<td>Timers</td>
<td>23</td>
</tr>
<tr>
<td>7.2</td>
<td>Threads</td>
<td>23</td>
</tr>
<tr>
<td>8.</td>
<td>An Overview of Networking MIDlets</td>
<td>26</td>
</tr>
<tr>
<td>9.</td>
<td>An Overview of Persistent Storage</td>
<td>27</td>
</tr>
<tr>
<td>9.1</td>
<td>Reading and Writing a Record's Data</td>
<td>28</td>
</tr>
<tr>
<td>10.</td>
<td>Nokia UI API</td>
<td>29</td>
</tr>
<tr>
<td>11.</td>
<td>Further Reading</td>
<td>30</td>
</tr>
</tbody>
</table>
Legal Notices

Copyright © Nokia Corporation 2002. All rights reserved.

Reproduction, transfer, distribution or storage of part or all of the contents in this document in any form without the prior written permission of Nokia is prohibited.

Nokia and Nokia Connecting People are registered trademarks of Nokia Corporation. Other product and company names mentioned herein may be trademarks or tradenames of their respective owners.

Nokia operates a policy of continuous development. Nokia reserves the right to make changes and improvements to any of the products described in this document without prior notice.

Under no circumstances shall Nokia be responsible for any loss of data or income or any special, incidental, consequential or indirect damages howsoever caused.

The contents of this document are provided "as is". Except as required by applicable law, no warranties of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose, are made in relation to the accuracy, reliability or contents of this document. Nokia reserves the right to revise this document or withdraw it at any time without prior notice.
# Glossary

The following terms and abbreviations are used within this document.

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>PNG</td>
<td>Portable Network Graphics format, version 1.0.</td>
</tr>
</tbody>
</table>
1. Introduction

1.1 Purpose and Scope

This document discusses issues involved in the design and implementation of a Series 60 Java™ application. The reader is expected to have read “Introduction to Series 60 Applications for Java Developers” and have good knowledge of Java™.
2. Overview

2.1 MIDlet Overview

A MIDlet is an application written for the Java 2 Micro Edition (J2ME™) Mobile Information Device Profile (MIDP).

The MIDP Specification (JSR-37), defines certain minimum hardware, software, networking, and application requirements. It also specifies standard system APIs for devices that support MIDP. MIDP is intended for use on "connected" devices with limited CPU, memory, keyboard, and display capabilities, such as cell phones and pager devices. The MIDP Specification also defines support for the HTTP client networking capabilities. MIDP implementations are required to support the PNG graphics format.

MIDP applications reside alongside other applications and functions on a mobile device. The MIDlet lifecycle is integral to enabling this co-existence. How MIDlets fit in with other applications and the host platform’s operating system is shown in Figure 1.

![Figure 1: High-level architecture of MIDP applications](image)

An understanding of several basic concepts is required when designing and implementing a MIDP application. These are: MIDlet lifecycle, user interface (using high- or low-level APIs), command handling, threads and timers, persistent storage, networking, and application management.

When designing MIDP applications for mobile phone use, developers should keep certain items in mind:

- The applications should be transparent to users who are not necessarily technology experts or familiar with devices other than the mobile phone they are currently using.
There are different kinds of MIDP devices, so developers need to consider different display sizes, different keyboards and the different "look and feel" of devices.

Developers should be aware that when the device receives a phone call or SMS during the execution of a MIDlet, the MIDlet loses the foreground. It should recover smoothly when it returns to the foreground.

The device could suddenly lose its network connection when a MIDlet is running. A networked application needs to be able to cope with such circumstances. Various test cases are crucial for developing high-quality Series 60 MIDP applications.

Series 60 Platform implementation of MIDP does not have a fixed amount of memory allocated for use by MIDlets; instead it shares the available memory dynamically with other types of applications. Queries as to the amount of memory available to a MIDlet, by using Runtime.getRuntime().freeMemory() or Runtime.getRuntime().totalMemory(), are therefore meaningless on a Series 60 device.

2.2 Basic Design Considerations

2.2.1 MIDlet limitations

This section takes a brief look at the basic limitations on MIDlets.

- MIDlets are unable to access the Series 60 device’s contacts or other native application data, although with networking, contacts could be stored on a server. This limits the range of applications possible.
- It is not possible to extend Java APIs beyond those that come with the device because MIDP has no Java Native Interface (JNI) support.
- Persistent data can be shared between MIDlets within the same suite. MIDlets cannot access the persistent data of MIDlets within other suites. MIDlets should be designed to consider this fact by grouping MIDlets that need to share data in the same suite.
- J2ME is essentially a subset of J2SE, which means that libraries (jar files) are not often easily transferable.
- Due to the limitations of MIDP devices, a good design in J2SE may require simplification for J2ME.

2.2.2 MIDlet UIs

When designing UIs for mobile phones, developers should be aware of the following issues:

- Target group. Designers should have a clear idea of their application’s target users before design and implementation begin.
- Early focus on users. Designers should have direct contact with the intended users to make sure their requirements are being fulfilled. Applications should be simple, easy, and comfortable to use.
- Early and continual user testing. Early testing is essential so that problems can be fed back into the development cycle as soon as possible.
- Integrated design. All aspects of usability (e.g., UI, help system, documentation) should evolve concurrently, rather than sequentially.
3. **Application Life Cycle**

The Application Management Software (AMS) is the environment in which a MIDlet is installed, started, stopped, and uninstalled. The AMS is sometimes referred to as the Java Application Manager (JAM). The AMS creates each new MIDlet instance and controls its state by directing a MIDlet to start, pause or destroy itself. The AMS is described in more detail in the MIDP Specification (JSR-37).

A MIDlet can be in one of the following states: Paused, Active, or Destroyed.

![Diagram of MIDlet lifecycle](image)

*Figure 2: The states of a MIDlet lifecycle*

The AMS constructs the MIDlet and communicates state changes to the MIDlet by calling methods that every MIDlet must implement. These are: `pauseApp()`, `startApp()`, and `destroyApp(boolean unconditional)`.

### 3.1 MIDlet Construction

A MIDlet's constructor is optional; typically it is used to construct and initialize objects. A MIDlet cannot access the display (Display object) from within its constructor.

### 3.2 StartApp()

The AMS calls `startApp()` to instruct the application that it has gained focus. This can occur when a MIDlet is newly constructed or when it has returned from a paused state. If the application has been newly constructed it will call `Display.getDisplay(this).setCurrent(startup_screen)`, to invoke its startup screen. If the application had been paused it will typically need to restore the state it was in when it was paused.

A flag is often used when implementing `startApp()` to test whether the MIDlet is returning from a paused state or has just been constructed.
3.3 PauseApp()

The AMS calls pauseApp() to instruct the application that it has lost focus. In this state the application cannot have any UI presence. Any threads or timers that the application has created are not automatically stopped. The application will need to terminate threads and cancel timers itself. The application will need to restore its state when it is reactivated.

It may be desirable to call Display.getDisplay(this).setCurrent(desired_screen), so that when the application is reactivated, the desired screen will be showing.

3.4 DestroyApp(boolean unconditional)

The AMS calls destroyApp() to instruct the MIDlet to release all the resources it has open, and to close down as soon as possible. This means the application should close all input and output streams, terminate all threads, and cancel all timers. Once destroyApp() has been called, the MIDlet cannot access the display.
4. UI Classes

4.1 Overview

The MIDP UI has a high-level and a low-level API. The high-level API’s classes: Alert, Form, List, and Textbox are extensions of the abstract class Screen. The low-level API is based on the use of the abstract class Canvas.

![Class diagram of MIDP screen types](image)

The high-level API classes are designed to provide abstractions and components that are highly portable across devices as the actual implementation takes care of aspects such as drawing, font characteristics, navigation, scrolling, etc. The particular device’s implementation of these classes performs the adaptation to its hardware and native UI look and feel.

The low-level API’s Canvas class allows applications to have more direct control of the UI. It allows greater control of what is drawn on the display and allows the receiving of low-level keyboard events. It is the application programmer’s responsibility to ensure portability across MIDs with different characteristics.

4.2 Navigation

A MIDlet can be seen as a set of screens through which the user navigates. The application is responsible for switching between these screens, which is achieved using the statement:

```java
Display.getDisplay(this).setCurrent(screen_to_show);
```

The navigational aspects of a MIDP application should be designed with care. Users need easy access to key features of the application, so navigation should not be too complicated, and entering data should be as simple as possible.
5. High-Level UI API

5.1 Command Handling

The Command object is used for controlling user interaction. Every Screen and Canvas can have an arbitrary number of Commands. Commands are mapped to real UI elements by the MIDP implementation. The Series 60 implementation of MIDP uses the Options menu for commands.

![Command example](image)

Figure 4: Series 60 command menu

The Command constructor has three parameters:

```
Command example = new Command (string, type, priority)
```

1. String – the string representation of the command that will be displayed.
2. Type – an indication to the device of the nature of the command (see next section).
3. Priority – the importance of the item.

5.2 Command Types

5.2.1 Exit command

The exit command type, as the name suggests, is used for commands that exit the application. The Series 60 MIDP implementation automatically adds an "Exit" option to the command menu. When the automatically added "Exit" is selected by the User the `destroyApp()` method of the MIDlet is called. There is no way to override this behavior. A developer specified "Exit" command would cause two "Exit" options to be added to the menu and potentially confuse the user. However, if the MIDlet needs to be portable, an "Exit" command must be specified. A developer therefore needs to weigh the need for portability against the potential confusion to the user of having two "Exit" options on the menu.

This behavior has some repercussions for application design. If a user chooses the "Exit" option from the menu, the MIDlet will not be able to display a screen requesting...
information from the user, for example a "Would you like to save the data?" message, because the screen is not accessible once `destroyApp()` has been called.

5.2.2 Screen command
The screen command type is used for commands that relate to a particular Screen. Typically, most application commands will be of this type.

5.2.3 Item command
The item command type is used for commands that are specific to the currently focused item on the screen.

5.2.4 Help command
The help command type is used for commands for help information relating to the current Displayable.

5.2.5 Back command
This command type is used for a command that navigates to the last display and is mapped to the right soft key.

5.2.6 OK/Cancel command
These command types are typically used for confirmation dialogs. "OK" is mapped to the left soft key and "Cancel" to the right soft key.

Command's application-level handling is based on a listener function. Each Displayable object has a single CommandListener. When the user invokes a Command on a Displayable, its listener is called. To define itself as a listener, an object must implement the interface CommandListener and its method, `commandAction()`. To register a CommandListener with a Displayable object, the Displayable's `setCommandListener()` method needs to be invoked.

It is possible for a single class to be a CommandListener for many Screens because the handleCommand() method is passed the Screen (Displayable) from which the Command was issued. Having a single class that handles all Commands is often favored in UI application design.

5.3 Designing Commands:

- Avoid using too many commands. Too many commands can make an application inconvenient to use as the user may need to scroll through the whole list to select a command.

- In the design phase, break the application into views and then design the interactions between each view. Screens and Commands are used to realize the actual tasks in views.

- Think carefully about the order of Commands in the menu. A basic principle is that the most commonly used functionality should come first in the list. Remember that "Exit" is always the last item in the menu and should always quit the application.

- Limit the length of the Command labels; describe the command as clearly and concisely as possible. Put the most important text at the start of the Command string because the device crops text that is greater than eight characters long. For example, if a developer has set a Command name as "Write greeting" the user actually sees on the screen: "Write...".
In Series 60 devices there are certain behavioral rules that will already be familiar to users. For example, normally, if there are two soft keys in the device (left and right) the right soft key provides "Back" or "Exit" functionality. To design easy to use applications for Series 60 devices, developers should respect these usability issues. In the MIDP implementation this is handled by Command mapping; "negative", "backward" actions are always mapped to the right soft key and "positive" actions to the left soft key.

5.4 The Screen Class

The Screen class is the common superclass of all high-level UI classes (see Figure 3). Every Screen object provides presentation and layout for itself. MIDP applications that use high-level UI components define only content. Screen classes map directly to native Series 60 UI controls.

The high-level UI components that are derived from Screen are: Alert, Form, List, and TextBox.

5.4.1 TextBox

A TextBox screen allows the user to enter or edit text. It has a title and text body.

Figure 5: A TextBox with no constraints (TextField.ANY)

TextBoxes can be customized for specific data entry requirements by setting their maximum text size and their constraints. Constraints tailor a TextBox for a specific type of data entry and are defined as static member variables in the class TextField.

TextBox constraints are:

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANY</td>
<td>Text has no constraints</td>
</tr>
<tr>
<td>EMAILADDR</td>
<td>E-mail address</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>Positive or negative integers</td>
</tr>
<tr>
<td>PASSWORD</td>
<td>The entered data will be obscured by stars</td>
</tr>
<tr>
<td>PHONENUMBER</td>
<td>For phone number entry</td>
</tr>
<tr>
<td>URL</td>
<td>For URL entry</td>
</tr>
</tbody>
</table>
Although an application can set the constraints for a TextBox, the application should still check the validity of entered data once the user has completed the screen.

5.4.2 List

The List class is a screen containing a scrollable list of choices. When a List is present on the display, the user can interact with it indefinitely. Scrolling operations on a List do not cause application events. A List is created by appending, inserting, or deleting items. A List item consists of text and an optional image.

The MIDlet can be notified of a user’s List selection via the CommandListener (see Command Handling, above), an extra Command type is defined by the List class (List.SELECT_COMMAND) for this purpose.

There are three different types of List (EXCLUSIVE, IMPLICIT, and MULTIPLE); these determine when the CommandListener gets notified of selection events relating to the List.

![Implicit ListBox](image)

**Figure 6: An implicit ListBox**

![Exclusive ListBox](image)

**Figure 7: An exclusive ListBox**
List types:

<table>
<thead>
<tr>
<th>List type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPLICIT</td>
<td>The listener is notified when the user selects an item from the list. Typically used as a menu screen.</td>
</tr>
<tr>
<td>EXCLUSIVE</td>
<td>The listener receives no notification of selection events. Typically a command will be added to the</td>
</tr>
<tr>
<td></td>
<td>List screen for users to indicate that they have completed their list selection.</td>
</tr>
<tr>
<td>MULTIPLE</td>
<td>The listener receives no notification of selection events. The select operation toggles the state of the</td>
</tr>
<tr>
<td></td>
<td>focused element. Typically a command will be added to the List screen for the user to indicate that</td>
</tr>
<tr>
<td></td>
<td>they have completed their list selection.</td>
</tr>
</tbody>
</table>

The item(s) selected by the user from the List can be retrieved using the getSelectedIndex() or getSelectedFlags() methods.

Example code for adding images to a list:

```java
Image starImage = null;
try {
    starImage = Image.createImage("test/star.png");
} catch (IOException e) {
    // image could not be found, starImage will be null and will therefore not appear.
}
myList.append("A Star", starImage);
```

An item consisting of an image and text is added to a List (myList). If createImage() throws an IOException (typically because the image file cannot be found), starImage will be null. List's append() method treats a null image parameter as a request that no image be shown.

The star image is located within the test folder in the MIDlet's JAR file.
5.4.3 Alert
Alerts are used to flash information to the user. Alerts consist of a title and text and/or an image. Alerts can be displayed for a specific duration of time or they can be modal, in which case they will be displayed until dismissed by the user.

![Error Alert Screen]

Figure 9: An error alert screen

A number of types of Alerts are available (defined in AlertType):

<table>
<thead>
<tr>
<th>Alert Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALERT</td>
<td>For events the user has previously requested</td>
</tr>
<tr>
<td>CONFIRMATION</td>
<td>For confirmation of user actions</td>
</tr>
<tr>
<td>ERROR</td>
<td>For error reporting</td>
</tr>
<tr>
<td>INFO</td>
<td>For displaying information</td>
</tr>
<tr>
<td>WARNING</td>
<td>For warning the user about a potentially dangerous situation</td>
</tr>
</tbody>
</table>

Although Alert extends Screen, it cannot have Commands or a CommandListener. Attempts to invoke Command methods on an Alert will result in an IllegalStateException being thrown.

When an Alert is displayed, the screen that appears after the Alert is dismissed must be specified; the Display class has a specific method for displaying Alerts:

```java
Display.getDisplay(this).setCurrent(alert_to_show, screen_to_show_after_dismissal)
```

5.4.4 Form
A Form is a screen to which items can be added. All items that can be added to a Form share the Item superclass. MIDP provides the following items: ChoiceGroup, DateField, Gauge, StringItem, and TextField.

A MIDlet receives notification of the changes to states of items contained on a Form via the ItemStateListener interface. A Form can have only one ItemStateListener, which will receive notification for all of the items contained on the Form. Notifications are handled...
by the ItemStateListener's method itemStateChanged(), which must be implemented when implementing the ItemStateListener interface.

5.4.4.1 ChoiceGroup

A ChoiceGroup is used for multiple- or single-option selection within a Form.

5.4.4.2 DateField

DateField is an editor for displaying or editing time and date. The ItemStateListener is notified when the user enters a new date/time.

5.4.4.3 Gauge

Gauge represents a numeric value (positive integers) in graphical form. There are two types of Gauge: interactive (modifiable by user) and non-interactive.
An interactive Gauge will increment/decrement its value in response to left and right key presses when it has focus. Typically a non-interactive Gauge is used as a progress bar, a timer or thread will update the Gauge periodically. The ItemStateListener is notified when the user modifies an interactive Gauge.

5.4.4.4 StringItem
StringItem is a non-editable text item.

5.4.4.5 TextField
TextField is an editable text component, it behaves in a similar manner to the TextBox screen component (see Section 5.4.1).

The ItemStateListener is notified when the TextField looses focus.

5.5 Practical User Interface Design Steps
1. A "screen map" or some other modeling technique should be used to visualize application design. Split the application into states and clarify connections between the states (an application has a starting point and an end point). Consider navigation throughout the application, ensure that users can always return to where they came from.
2. The notification messages required need to be planned. Error cases need to be considered and made part of the application testing process.

3. Take advantage of the MIDP high-level UI where possible, so a MIDlet behaves like the phone's own applications.

4. Remember to define and implement comprehensive help within applications so that users can immediately find answers to their questions.
6. Low-Level UI API

Canvas is the base class for classes that require low-level control of event-handling and graphical presentation. A Canvas derived object can be interchanged with any of the Screen classes (Alert, Form, Textbox, or List) because they all implement the Displayable interface (see Figure 3). Canvas is an abstract base class, which means it must be extended; the paint(Graphics g) function must be implemented. Like the Screen classes, Canvas also has Commands.

6.1 Low-Level Event Handling with Canvas

In MIDP, events are delivered to the Canvas using the following set of notification methods:

- showNotify()
- hideNotify()
- keyPressed()
- keyRepeated()
- keyReleased()

A developer implements the notification methods as required. Event notification methods are called serially by the MIDlet framework; this means that until a notification method has completed, no other notification methods will be called. It is therefore best to avoid long-running tasks within event notification methods, such tasks should be done in a separate thread or using a timer.

6.1.1 ShowNotify()

showNotify() is called to notify the Canvas it is about to be made visible.

6.1.2 HideNotify()

This is called to notify the Canvas it is about to be hidden.

6.1.3 KeyPressed(), keyRepeated(), and keyReleased()

The key code is passed into each of these methods as a parameter. Typically games will need to implement these key event methods. The MIDP Specification defines a set of "Game Actions"; these remove the burden from game developers of mapping keys to game actions. The getGameAction() returns the game action of the key code. Game actions are defined as static member data of the Canvas class, e.g., LEFT, RIGHT, UP, DOWN, GAME_A, etc.

An example of how to use "Game Actions":

```java
protected void keyPressed(int keyCode)
{
    int gameAction = getGameAction(keyCode);
    switch (gameAction)
    {
        case FIRE:
            doSomething();
            break;
        case RIGHT:
            doSomethingElse();
            break;
    }
}
```
default:
    break;
}
}

6.2 Canvas Graphics

When a Canvas object is required to draw itself on the screen, its paint() method is called. A Graphics object is passed into the paint() method; this provides a basic API for drawing simple shapes to the screen (see Section 6.3, Graphics, for more details). The graphics object is only valid for the duration of the paint() method.

An application can request a repaint of a Canvas by calling the Canvas object’s repaint() method; the repaint request is not immediate. To force a repaint, the application needs to call the serviceRepaints() method after calling repaint(). The serviceRepaints() method blocks until the Canvas has been repainted.

6.3 Graphics

The Graphics object represents a graphics context through which drawing can take place. A graphics context has a destination, which is either the actual device screen or an off-screen image. Off-screen images are often used to implement double buffering. Graphics object can be obtained calling the getGraphics() method. The object of an off-screen image can be held indefinitely by the MIDlet.

A Graphics object provides a set of methods for rendering drawing primitives: text, images, lines, rectangles, and arcs; it also supports clipping and origin translation methods.

A Graphics context has a color associated with it, which it will use when rendering primitives. MIDP supports a 24-bit color model (red, green and blue have 8 bits each). A device's color characteristics are obtained using the Display object's numColors() method.

6.3.1 Displaying text

The Font class provides a set of methods for determining the height and width of text when it is displayed on the screen. The Font class also provides static methods for querying the device about available fonts.

6.3.2 Image class

The Image class provides several static methods for the creation of images: from a file (PNG), from another image, and from a raw byte array. Images are either mutable (editable) or immutable (fixed). An image created from a file is immutable.

A graphical context can be obtained for a mutable image but an immutable image will throw an exception if the getGraphics() method is called. The Image class provides a method for creating a mutable image from an immutable image:

Image.createImage(immutable_image)
7. **Timers and Threads**

7.1 **Timers**

Timers enable applications to schedule tasks easily without resorting to the use of threads. Using threads usually requires a more complex design and can require more system resources, which are at a premium on small devices.

An example of the creation of a timer:

```java
public class MyTask extends TimerTask {
    private int iCount;
    public MyTask(int aStart) {
        super();
        iCount = aStart;
    }

    public void run() {
        iCounter--;
        System.out.println("Counter is now " + iCounter);
        if (iCounter == 0)
            cancel();
    }
}
```

To construct a timer and add MyTask to it:

```java
MyTask myTask = new MyTask(50);
Timer myTimer = Timer();
myTimer.schedule(myTask, 1000, 20000);
```

In the example, the run() method of MyTask is called every 20 seconds.

Timer tasks are not guaranteed to be executed at the time requested. Timer tasks are run one after another. If one task takes a long time to complete, the next task will be executed only after the current task has completed. For repeated tasks it is best to ensure that the run() method completes quickly.

It is possible to have more than one timer and then assign tasks to both. However, care should be taken when using multiple timers because each timer runs in its own thread; therefore, synchronization may be required.

7.2 **Threads**

MIDP supports a reduced API version of the J2SE's java.lang.Thread class (the interrupt() and stop() methods are not supported in J2ME). Threads are commonly used to execute input/output operations and background tasks. A thread is not halted or notified by the AMS when the application changes to a paused or destroyed state; this is the task of the application developer.
On rare occasions, a MIDlet may need background threads running while it is in a "paused" state. Typically a MIDlet should terminate threads when pauseApp() is called and restart them when the MIDlet is activated again.

After destruction of a MIDlet, the AMS will cleanup any threads that an application still has running. It is not desirable to rely on this behavior. MIDlets should always try to close resources and terminate threads cleanly before destroyApp() completes.

An example of using a thread:

```java
class TestThread extends Thread {
    boolean running=true;

    public void run() {
        while (running) {
            try {
                doSomething();
                synchronized (this) {
                    wait(1000);
                }
            } catch (InterruptedException e) {
            }
        }
    }

    public void kill() {
        synchronized (this) {
            notify();
            running = false;
        }
    }
}
```

Below is an example of how a MIDlet could implement destroyApp() to ensure that its TestThread object (myThread) is terminated as cleanly as possible:

```java
public void destroyApp(boolean unconditional) {
    myThread.kill();
    try {
        // wait for the thread to actually die before continuing
        thread.join();
    } catch (InterruptedException e) {
        // InterruptedException is never thrown
    }
}
```
The destroyApp() method requests that the thread terminates itself (using myThread.kill()). It then waits for the thread to complete (using myThread.join()). In this way the destroyApp() method ensures the thread has terminated before it completes.
8. An Overview of Networking MIDlets

The CLDC 1.0 Specification defines an abstract Generic Connection Framework (GCF) for MIDP networking. This framework defines the form of the networking APIs but not any actual protocol implementations.

The framework includes support for
- Stream-based connections
- Datagrams

The MIDP Specification defines that a compliant implementation must implement at least HTTP1.1 stream-based client connections (RFC2616), using the CLDC Generic Connection Framework (GCF). A fully portable MIDlet should not use more than the HTTP networking APIs defined in the MIDP Specification.

The GCF defines a Connector class, which has a set of static factory methods (open()) that return a Connection object. The Connector returns a Connection object appropriate to the protocol used in the open() method.

```java
HttpConnection connection = (HttpConnection)Connector.open(
    "http://www.sun.com", Connector.READ_WRITE);
```

In the example above the Connector returns an HttpConnection object.

The HttpConnection class provides an API for standard HTTP networking; included are methods for getting HTTP header information. HttpConnection also provides: openInputStream(), openDataInputStream(), openOutputStream(), and openDataOutputStream(). These methods can be used to send and receive data.

Networking operations are often slow and are therefore typically initiated in a separate thread.
9. An Overview of Persistent Storage

MIDlets often need to store/retrieve data; this could be:

- Preference information
- A history of the user's past inputs
- Application data, for example a Notepad MIDlet

The MIDP Specification provides a persistence mechanism for MIDlets: the Record Management System package (javax.microedition.rms). At its core is the RecordStore class. A record store is a collection of records that persists across multiple invocations of a MIDlet and are shared with other MIDlets within the same suite.

A MIDlet suite can have multiple RecordStores, each identified by a string. Each of the records that comprise a RecordStore is identified by a unique ID.

The RecordStore class provides a set of static methods for managing the RecordStores within a MIDlet suite:

- listRecordStores() which returns an array of the names of all the RecordStores that the MIDlet suite contains.
- openRecordStore() which returns a RecordStore object for the requested record store.
- deleteRecordStore() which deletes the requested RecordStore.

A RecordStore object has methods for

- Adding, deleting, retrieving and modifying records
- Enumerating through the records within a RecordStore
- Setting a RecordListener for objects, which require notification of record addition, deletion and/or modification
- Closing

The data in a record store is stored in chunks called records. A record is a set of arbitrary binary data - an array of bytes. An entire record must be read or written. RecordStore provides the following methods for this purpose: getRecord(), addRecord(), and setRecord().

An example showing the typical pattern of RecordStore access:

```java
RecordStore rs = null;
byte[] data;
int id = 2; // record ID of record required
try {
   rs = RecordStore.openRecordStore("myrecordstore", false);
   data = rs.getRecord(id);
   ...
   // modify the data object
   rs.setRecord(id, data, 0, data.length);
   rs.closeRecordStore();
}

catch( Exception e )
{
}
```
... // do something here
}

The code opens a RecordStore named “myrecordstore”. It retrieves a record, modifies the record, sets the modified record data, and finally closes the RecordStore.

9.1 Reading and Writing a Record’s Data

As stated above, the data that a record contains is stored in a byte array. J2ME, unlike J2SE, does not support object serialization; the MIDlet must serialize objects itself. A method for achieving this is demonstrated in the example below.

An Example showing writing data to a byte array:

```java
ByteArrayOutputStream bos = new ByteArrayOutputStream();
DataOutputStream out = new DataOutputStream(bos);
out.writeInt(40);
out.writeBoolean(true);
out.writeUTF("A test string");
data = bos.toByteArray();
out.close();
```

The ByteArrayOutputStream class provides a toByteArray() method for converting a stream into a byte array. The DataOutputStream class provides a set of methods for writing Java primitives (int, long, Boolean, String, etc.) to a stream. The data can be read back from the byte array using the corresponding input stream classes and methods.
10. Nokia UI API

Nokia's UI API is an extension of the standard MIDP1.0 APIs and is available in Series 60 devices. As MIDP was designed to be as portable as possible, certain useful capabilities, such as sound and graphics transparency, were excluded as they couldn't be implemented by all potential MIDP devices. Nokia's UI API was created to make some of these useful features available.

The API consists of four classes and two interfaces in two packages as shown in the table below:

<table>
<thead>
<tr>
<th>Package</th>
<th>Classes</th>
<th>Interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.nokia.mid.ui</td>
<td>FullCanvas, DirectUtils, DeviceControl</td>
<td>DirectGraphics</td>
</tr>
<tr>
<td>com.nokia.mid.sound</td>
<td>Sound</td>
<td>SoundListener</td>
</tr>
</tbody>
</table>

The extensions fall into three categories: UI extensions, sound extensions and, vibration and screen backlight control. The UI extensions are in the com.nokia.mid.ui package and add the following features: a full-screen Canvas class, drawing and filling triangles and polygons, drawing images reflected or rotated, transparency support (making non-rectangular sprites possible), extra ways to create mutable images, and low-level access to image pixel data. The sound extensions are in the com.nokia.mid.sound package and add the ability to play single notes and simple tunes, and control their volume. The vibration and screen backlight control extensions are in the com.nokia.mid.ui package and add the ability to start and stop the phone's vibration feature, switch the backlight on and off, and flash the phone's lights.

For example, to play a single note:
```java
Sound s = new Sound(440, 100);
s.play(1);
```

The parameters to the Sound constructor are frequency (in Hertz) and duration (in milliseconds). The parameter to play is how many times to repeat the sound.

To start the phone's vibration feature:
```java
DeviceControl.startVibra(100, 500);
```

The parameters to startVibra are frequency (not in Hertz but in the range 0 to 100) and duration (in milliseconds).

For detailed information about how to use Nokia's UI API see "The Nokia UI API Programmer's Guide" available from the Forum Nokia website.
11. Further Reading

- Mobile Information Device Profile (JSR-37), http://java.sun.com/products/midp/

Books
- J2ME In a Nutshell (O'REILLY)