# Table of Contents

1. **Introduction** ................................................................................................................ 7
   1.1 Purpose and Scope ........................................................................................................... 7
   1.2 Developer Platform 2.0 for Series 60 ............................................................................. 7
      1.2.1 MIDP 2.0 .................................................................................................................. 7

2. **Overview** ..................................................................................................................... 8
   2.1 MIDlet Overview ............................................................................................................. 8
   2.2 Basic Design Considerations .......................................................................................... 9
      2.2.1 MIDlet limitations .................................................................................................... 9
      2.2.2 MIDlet UIs ............................................................................................................... 9

3. **Application Life Cycle** ................................................................................................ 9
   3.1 MIDlet Construction ........................................................................................................ 10
   3.2 `startApp()` Method ..................................................................................................... 10
   3.3 `pauseApp()` Method ................................................................................................... 10
   3.4 `destroyApp (boolean unconditional)` Method ............................................................. 11

4. **UI Classes** .................................................................................................................. 12
   4.1 Overview ....................................................................................................................... 12
   4.2 Navigation ..................................................................................................................... 12

5. **High-Level UI API** ..................................................................................................... 13
   5.1 Command Handling ....................................................................................................... 13
   5.2 Command Types ............................................................................................................ 13
      5.2.1 EXIT command ........................................................................................................ 13
      5.2.2 SCREEN command ................................................................................................. 14
      5.2.3 ITEM command ....................................................................................................... 14
      5.2.4 HELP command ...................................................................................................... 14
      5.2.5 BACK command ..................................................................................................... 14
      5.2.6 OK/CANCEL commands ......................................................................................... 14
      5.2.7 STOP command ....................................................................................................... 14
   5.3 Designing Commands .................................................................................................... 14
   5.4 The Screen Class .......................................................................................................... 15
      5.4.1 TextBox .................................................................................................................. 15
      5.4.2 List ........................................................................................................................ 16
      5.4.3 Alert ....................................................................................................................... 18
      5.4.4 Form ....................................................................................................................... 19
      5.4.4.1 ChoiceGroup ........................................................................................................ 19
      5.4.4.2 CustomItem ......................................................................................................... 20
5.4.4.3 DateField..................................................................................................................20
5.4.4.4 Gauge....................................................................................................................20
5.4.4.5 StringItem.............................................................................................................20
5.4.4.6 TextField............................................................................................................21

6. Low-Level UI API ........................................................................................................ 22
6.1 Low-Level Event Handling with Canvas .................................................................. 22
6.1.1 hideNotify() .........................................................................................................23
6.1.2 showNotify() .......................................................................................................23
6.1.3 keyPressed(), keyRepeated(), and keyReleased() ..............................................23
6.1.4 MIDP 2.0 GameCanvas key detection ...................................................................24

6.2 Canvas Graphics ........................................................................................................ 25
6.2.1 MIDP 2.0 GameCanvas Graphics .........................................................................25

6.3 Graphics .................................................................................................................. 25
6.3.1 Displaying text ......................................................................................................25
6.3.2 Image class ..........................................................................................................26

7. Timers and Threads ..................................................................................................... 27
7.1 Timers .......................................................................................................................27
7.2 Threads .....................................................................................................................27

8. An Overview of Networking MIDlets ......................................................................... 29
8.1 Generic Connection Framework (GCF) ....................................................................29
8.2 Wireless Messaging API (JSR-120) .........................................................................30
8.2.1 The base Message class ......................................................................................30
8.2.2 Sending and receiving messages .........................................................................30
8.3 Bluetooth API (JSR-82) ..........................................................................................31

9. An Overview of Persistent Storage ............................................................................. 35
9.1 Reading and Writing a Record's Data .......................................................................36

10. Trusted and Untrusted MIDlet Suites ..................................................................... 37
10.1 Untrusted MIDlet Suites .......................................................................................37
10.2 Trusted MIDlet Suites ...........................................................................................37
10.3 Protection Domains for Trusted MIDlet Suites .....................................................37
10.4 Installation ...............................................................................................................38

11. Nokia UI API ............................................................................................................. 39
11.1 MIDP 2.0 and Nokia UI API ..................................................................................39

12. An Overview of Mobile Media API (JSR-135) ........................................................ 41
12.1 Media Processing ....................................................................................................41
12.2 Developer Platform 2.0 for Series 60 MMAPI Implementation ...............................43
13. An Overview of Push Registry ................................................................. 44
14. Further Reading ................................................................................ 46
Legal Notices

Copyright © Nokia Corporation 2003. 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 trade names 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>AMR</td>
<td>Adaptive Multi Rate</td>
</tr>
<tr>
<td>AMS</td>
<td>Application Management Software</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CLDC</td>
<td>Connected Limited Device Configuration</td>
</tr>
<tr>
<td>DRM</td>
<td>Digital Right Management</td>
</tr>
<tr>
<td>GCF</td>
<td>Generic Connection Framework</td>
</tr>
<tr>
<td>J2ME™</td>
<td>Java 2 Platform, Micro Edition</td>
</tr>
<tr>
<td>J2SE™</td>
<td>Java 2 Platform, Standard Edition</td>
</tr>
<tr>
<td>JAD</td>
<td>Java Application Description</td>
</tr>
<tr>
<td>JAM</td>
<td>Java Application Management</td>
</tr>
<tr>
<td>JAR</td>
<td>Java Archive</td>
</tr>
<tr>
<td>JSR</td>
<td>Java Specification Request</td>
</tr>
<tr>
<td>MIDI</td>
<td>Musical Instrument Digital Interface</td>
</tr>
<tr>
<td>MIDP</td>
<td>Mobile Information Device Profile</td>
</tr>
<tr>
<td>OBEX</td>
<td>Object Exchange Protocol</td>
</tr>
<tr>
<td>OTA</td>
<td>Over the Air</td>
</tr>
<tr>
<td>PNG</td>
<td>Portable Network Graphics format, version 1.0.</td>
</tr>
<tr>
<td>SMS</td>
<td>Short Message Service</td>
</tr>
<tr>
<td>SP-MIDI</td>
<td>Scalable Polyphony MIDI</td>
</tr>
<tr>
<td>URI</td>
<td>Uniform Resource Identifier</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
</tbody>
</table>
1. Introduction

1.1 Purpose and Scope

This document discusses issues involved in the design and implementation of a Java™ application for Developer Platform 2.0 for Series 60. It is an update to the older document, Designing Java™ Applications for Series 60. The reader is expected to have read Introduction to Series 60 Applications for Java™ Developers and have good knowledge of Java technology.

1.2 Developer Platform 2.0 for Series 60

Developer Platform 2.0 for Series 60 supports the installing and running of Java applications. Supported Java technology is the Java 2 Platform, Micro Edition (J2ME™), consisting of Mobile Information Device Profile (MIDP) 2.0 (JSR-118) and Connected Limited Device Configuration (CLDC) 1.0 with CLDC HotSpot virtual machine implementation, which are delivered as part of Symbian OS v7.0s. For more information about the J2ME platform and related specifications, see http://java.sun.com/j2me/.

In addition, the following Java APIs are supported:

- **Java Wireless Messaging API (JSR-120)** – Allows sending and receiving of SMS messages in text and binary formats.
- **Java Mobile Media API (JSR-135)** – Provides access and control of basic audio and multimedia features.
- **Java Bluetooth API (JSR-82)** – Enables Java applications to use Bluetooth connectivity (http://jcp.org/jsr/detail/82.jsp).
- **Nokia UI API** – Nokia proprietary API enables Java applications to use sound, full screen graphics (FullCanvas class), etc.

1.2.1 MIDP 2.0

Developer Platform 2.0 for Series 60 supports all of the mandatory features of MIDP 2.0. MIDP 2.0 Security Recommended Practices is partially supported. The following security domains are supported as specified in the MIDP 2.0 specification/security addendum: "Manufacturer domain, trusted third-party domain, and untrusted domain."

The implementation for the javax.microedition.media part of MIDP 2.0 includes support for the audio formats listed in this document in Chapter 12. It also includes support for vibrate(int) and flashBacklight(int) methods in the platform, provided that the device in question has support for vibra.

The following I/O interfaces are supported:

- HttpConnection
- HttpsConnection
- SecureConnection
- ServerSocketConnection
- SocketConnection (TCP)
- UDPPacketConnection
2. Overview

2.1 MIDlet Overview

A MIDlet is an application written for 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 mobile phones and pager devices. The MIDP specification also defines support for 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 coexistence. Figure 1 illustrates how MIDlets fit in with other applications and the host platform’s operating system.

Figure 1: High-level architecture of MIDP applications

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 Short Message Service (SMS) message 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 heap memory allocated for use by MIDlets; instead it shares the available memory dynamically with other types of applications. Queries as to the amount of heap 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 of 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.

- 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 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 1.0 specification (JSR-37).

A MIDlet can be in one of the following states: paused, active, or destroyed, as shown in Figure 2.

Figure 2: Life cycle of a MIDlet

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). The MIDlet starts out at the paused state, after which JAM automatically calls the MIDlet’s startApp() method and the MIDlet’s state changes to the active state.

3.1 MIDlet Construction

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

3.2 startApp() Method

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 within the startApp() method to test whether the MIDlet is returning from a paused state or has just been constructed.

3.3 pauseApp() Method

When the AMS calls pauseApp(), it instructs the application that it has lost focus. In this state, the application does not have any UI presence. Any threads or timers that the application has
created continue running unless manually stopped. The application will need to terminate threads and cancel timers itself. The application will need to restore its state when it is reactivated.

In Nokia implementations, pauseApp() is never called. Appropriate methods can be called, but the implementation does nothing (see Figure 2: Life cycle of a MIDlet).

3.4 destroyApp (boolean unconditional) Method

The AMS calls the destroyApp() method to instruct the MIDlet to release all the resources it has reserved, 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 UI API 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 2.0 screen types](diagram.png)

The high-level UI 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 UI 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 different devices 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 the 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.

The Command constructor has three parameters:

\[
\text{Command example} = \text{new Command (string, type, priority)}
\]

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 this 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.
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
The Back 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 commands
These command types are typically used for confirmation dialogs. OK is mapped to the left soft key and Cancel to the right soft key.

5.2.7 STOP command
The Stop command type is generally used for stopping a currently running process or operation.

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.

There are certain rules concerning how Commands are mapped in Series 60 devices. In brief, the following command types are mapped to the left soft key: SCREEN, OK, HELP and ITEM. Correspondingly, BACK, CANCEL, STOP and EXIT command types are mapped to the right soft key.

In MIDP 2.0 implementations, Commands can also be mapped to Items, in addition to the Displayables. Commands should be added to an Item, and ItemCommandListener should be set for the Command.

5.3 Designing Commands
- Avoid using too many commands. It can make an application inconvenient to use as the user may need to scroll through an entire 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. The most commonly used functionality should come first in the list. 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,” what the user actually sees on the screen is: "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.

In Series 60 MIDP implementations there is always an automatically created Options menu in the left soft key where there is an Exit command for closing the MIDlet.

5.4 The Screen Class

The Screen class is the common super class 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. In Series 60 MIDP 2.0 implementations, if the TextBox type is PHONENUMBER, the user can open the contacts database to retrieve a phone number.

![TextBox](image)

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>ANY</th>
<th>Text has no constraints.</th>
</tr>
</thead>
<tbody>
<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>Phone number entry.</td>
</tr>
<tr>
<td>URL</td>
<td>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.

In MIDP 2.0 there are also modifier flags for TextBox and TextField classes.

<table>
<thead>
<tr>
<th>PASSWORD</th>
<th>The text entered is confidential data that should be obscured.</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNEDITABLE</td>
<td>Editing is currently disallowed.</td>
</tr>
<tr>
<td>SENSITIVE</td>
<td>The text entered is sensitive data that the implementation must never store into other input schemes.</td>
</tr>
<tr>
<td>NON_PREDICTIVE</td>
<td>The text entered does not consist of words that are likely to be found in dictionaries.</td>
</tr>
<tr>
<td>INITIAL_CAPS_WORD</td>
<td>A hint to the implementation that during text editing, the initial letter of each word should be capitalized.</td>
</tr>
<tr>
<td>INITIAL_CAPS_SENTENCE</td>
<td>A hint to the implementation that during text editing, the initial letter of each sentence should be capitalized.</td>
</tr>
</tbody>
</table>

### 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 will be notified of selection events relating to the List.
List types:

| IMPLICIT   | The listener is notified when the user selects an item from the list. Typically used as a menu screen. |

Figure 6: An implicit ListBox

Figure 7: An exclusive ListBox

Figure 8: A multiple ListBox
### EXCLUSIVE

The listener receives no notification of selection events. Typically a command will be added to the List screen for users to indicate that they have completed their list selection.

### MULTIPLE

The listener receives no notification of selection events. The select operation toggles the state of the focused element. Typically a command will be added to the List screen for users to indicate that they have completed their list selection.

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.

![Alert Image](image-url)

Figure 9: An error Alert screen
A number of types of Alerts are available (defined in AlertType):

<table>
<thead>
<tr>
<th>ALERT</th>
<th>For events the user has previously requested.</th>
</tr>
</thead>
<tbody>
<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:

```
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 super class. MIDP 2.0 provides the following items: ChoiceGroup, CustomItem, DateField, Gauge, ImageItem, Spacer, 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.

---

**Figure 10: ChoiceGroups (exclusive and multiple)**
A ChoiceGroup can be either "exclusive" or "multiple" (see List types above). The ItemStateChangedListener is notified when the user modifies the choice made.

5.4.4.2 CustomItem

CustomItem is a customizable component for Forms, which can have complex visual and interactive layouts created by using different standard elements.

![CustomItem](image)

Figure 11: CustomItem with a TextField and a customized table

5.4.4.3 DateField

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

![DateField](image)

Figure 12: DateField (date) and DateField (date and time)

5.4.4.4 Gauge

Gauge represents a numeric value (positive integers) in graphical form. There are two types of Gauge: interactive (modifiable by user) and non-interactive.

![Gauge](image)

Figure 13: An interactive Gauge and a non-interactive Gauge

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 ItemStateChangedListener is notified when the user modifies an interactive Gauge.

5.4.4.5 StringItem

StringItem is a non-editable text item.
5.4.4.6 TextField

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

![TextField](image)

Figure 14: A form with a phone TextField and a password TextField

The ItemStateListener is notified when the TextField loses focus.
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 15). 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. A new feature in MIDP 2.0 is the new class GameCanvas, which greatly increases a developer’s possibilities for creating attractive games with animated graphics.

Figure 15: Class diagram of GameCanvas and related classes

6.1 Low-Level Event Handling with Canvas

In both MIDP 1.0 and MIDP 2.0, a virtual machine is able to trigger various events on Canvas, such as key presses, releases, repeats, and whether Canvas is still visible or not. The following five methods can be implemented in each Canvas class:

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

Developers implement 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 hideNotify()

hideNotify() is called right after the Canvas has left the display. It is extremely important to implement an auto-pause functionality here, as the Series 60 Platform's multitasking OS continues running the MIDlet in the background. Auto-pause is implemented by stopping threads, canceling timers, saving important values, etc., in the hideNotify() method.

```java
protected void hideNotify()
{
    remainingTime = endTime - System.currentTimeMillis();
    myThread.stop();
    autoPaused = true;
    repaint();
    // include a pause test in paint() method to check if paused
    // paint a pause message on screen if autoPaused true
}
```

```java
protected void paint(Graphics g)
{
    // paint game screen here
    if (autoPaused == true) {
        // paint pause message
    }
}
```

6.1.2 showNotify()

showNotify() is called just before the Canvas is getting back on display. In this method it is useful to restore important values, timers, restart threads, etc., so that application execution can continue as normal.

```java
protected void showNotify()
{
    myThread = new Thread(this);
    // include a pause test in keyPressed() method for continue
    // set new endTime and set autoPaused false, then repaint
}
```

```java
protected void keyPressed(int keyCode)
{
    if (autoPaused == true) {
        autoPaused = false;
        endTime = System.currentTimeMillis() + remainingTime;
        myThread.start();
        repaint();
    }
    // rest of key detection routine here
}
```

6.1.3 keyPressed(), keyRepeated(), and keyReleased()

These methods are called whenever something happens with the phone keypad. Upon a key press, keyPressed() is called, and when a key is released, the keyReleased() method is called. If a key is pressed for a period of time (approximately half a second in Nokia phones), the keyRepeated() method is called instead. The key code of the key is passed into each of these methods as a parameter. The MIDP specification defines a set of
Game Actions; these remove the burden from game developers of mapping keys to game actions. The getGameAction() method 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., which are mapped accordingly for each of the different devices, regardless of the physical layout.

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

### 6.1.4 MIDP 2.0 GameCanvas key detection

MIDP 2.0 introduces a new mechanism for detecting key events on low-level UI classes. With GameCanvas, it is possible to check which keys have been pressed since the last check. Key-press status is reset to zero after each check. The following example points out how to implement key detection in a simple game loop running on a thread:

```java
public void run() {
    Thread currentThread = Thread.currentThread();
    try {
        while (currentThread == myThread) {
            long startT = System.currentTimeMillis();
            int keyState = getKeyStates();
            if ((keyState & LEFT_PRESSED != 0) {
                sprite.move(-1, 0);
            }
            else if ((keyState & RIGHT_PRESSED != 0) {
                sprite.move(1, 0);
            }
            else if ((keyState & UP_PRESSED != 0) {
                sprite.move(0, -1);
            }
            else if ((keyState & DOWN_PRESSED != 0) {
                sprite.move(1, 0);
            }
            g.setColor(0x000000);
            g.fillRect(0, 0, getWidth(), getHeight());
            sprite.paint(g);
            flushGraphics();
            long tTaken = System.currentTimeMillis() – startT;
            if (tTaken < 40) {
                synchronized(this) {
                    wait(40 – tTaken);
                }
            }
        }
    }
}
else {
    wait(1);
}

} catch (InterruptedException e) {
    }

In the example, keys are monitored every time within a 40-millisecond loop and sprite is redrawn every time. For optimizing, redrawing can be left out if nothing has changed.

### 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.

#### 6.2.1 MIDP 2.0 GameCanvas Graphics

For MIDP 2.0, repainting is made simpler. The Graphics instance acquired by getGraphics() method is actually an off-screen image with size matching width and height of the current Canvas. Game screen is then painted on this Graphics object as normal and finally flushed on screen with flushGraphics() method. Developer doesn't need to call repaint() and serviceRepaints() manually any more. This can be seen above in the game loop example.

### 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 objects can be obtained by 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:

\[ \text{Image.createImage(immutable\_image)} \]
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 clean up 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 terminate itself (using myThread.kill()). It then waits for the thread to complete (using myThread.join()). In this way, the destroyApp() method ensures that the thread has terminated before it completes.
8. An Overview of Networking MIDlets

8.1 Generic Connection Framework (GCF)

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 and Datagrams, as shown in Figure 16.

![Diagram of CLDC Generic Connection Framework Interfaces and MIDP 2.0 Interfaces](image)

**Figure 16: Relationship of MIDP 2.0 Datagram and Socket interfaces to CLDC's Generic Connection Framework**

The MIDP 1.0 specification states 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 returns 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.
The MIDP 2.0 specification states that a compliant implementation must also implement a secured HTTP connection, HTTPS. And MIDP 2.0 provides a number of interfaces intended to simplify UDP-, TCP-, and TLS-based communication. A UDPDatagramConnection is used to send or receive UDP datagram messages. A ServerSocketConnection is used to accept inbound TCP connection requests. A SocketConnection is created for each accepted inbound TCP socket connection. A SocketConnection is also used for outbound TCP connections. Use of a SecureConnection is somewhat similar to use of a SocketConnection, except for the security considerations (e.g., TLS is used instead of TCP).

```java
SocketConnection socketConnection = (SocketConnection)Connector.open("socket://myhost:6006");
```

In the example above, the Connector returns a SocketConnection object. Networking operations are often slow and are therefore typically initiated in a separate thread.

### 8.2 Wireless Messaging API (JSR-120)

Wireless Messaging API (WMA) is based on the GCF defined in CLDC 1.0 and is an extension to the Java MIDP environment. It is derived from the Java Community Process, JSR-120, and specifies mappings for GSM SMS, GSM Cell Broadcast Service, and CDMA SMS Messages. WMA is intended to be extensible in the future to cover other messaging protocols, such as 3GPP MMS, SIP, and Instant Messaging.

WMA allows text messages to be sent, as well as text and binary messages to port numbers in a MIDlet or other native application. Messages also can be received by the MIDlet, using a port number.

The interfaces are different from the usual input/output networking functionality in J2ME profiles. Although similar in functionality, it was thought too confusing to derive the messaging API from the datagram. Therefore, WMA has been defined in the javax.wireless.messaging.

#### 8.2.1 The base Message class

The base class and representation of a message is called Message. Methods for addresses and time stamps are provided in this class. Both text and binary messages have been accommodated with two subinterfaces of Message —TextMessage and BinaryMessage —that allow the developer to manipulate the payloads of the message as Strings and byte arrays, respectively.

#### 8.2.2 Sending and receiving messages

The MessageListener and MessageConnection interfaces are the most essential classes in the WMA.

The MessageListener interface is implemented by the application to facilitate the notification of incoming messages. When an incoming message arrives, the notifyIncomingMessage() method is called. The application can then retrieve the message using the receive() method of the MessageConnection.

The MessageConnection interface defines the basic functionality for sending and receiving messages.

The application obtains an object implementing the MessageConnection from the Connector interface by providing a URL-like string that identifies an address.
If the application specifies a full destination address to the Connector, it creates a MessageConnection, which works in "client" mode and is used for sending messages. The Connection can only be used for sending messages to the address specified when it is instantiated. Conversely, the application can also operate in "server" mode and be used for receiving messages.

The MessageConnection object is created by the provision of the URL connection string, which is made up of two parts: the Address and the Data. The Address specifies the phone number and the port to which a MIDlet listening on the same port number on the recipient's device will receive the message. The string looks like this:

```java
String addr = "sms://+358401234567:5432";
```

The MIDlet listening on the port specified above will need to specify the same port number when it operates in server mode. The address string is constructed as follows:

```java
String addr = "sms://:5432";
```

The data to be sent is the payload of the message. The example below illustrates how message can be sent:

```java
try {
    String addr = "sms://+358401234567";
    MessageConnection conn = (MessageConnection)Connector.open(addr);
    TextMessage msg = (TextMessage)conn.newMessage(
        MessageConnection.TEXT_MESSAGE);
    msg.setPayloadText("Hello World!");
    conn.send(msg);
} catch (Exception e) {} 
```

8.3 Bluetooth API (JSR-82)

Bluetooth is a short-range universal wireless connectivity standard for electronic appliances and mobile devices. Java Bluetooth API (JSR-82) is a standardized set of Java APIs that allows Java-enabled devices to integrate into a Bluetooth environment.

JSR-82 is based on version 1.1 of the Bluetooth specification. It hides the complexity of the Bluetooth protocol stack behind a set of Java APIs that allows developers to focus on application development rather than the low-level details of Bluetooth.

The JSR-82 specification defines two packages:

- `javax.bluetooth`
- `javax.obex`

The first package is the core Bluetooth API and the second package is the API for OBEX. The OBEX API is independent from the Bluetooth API; therefore both APIs can be implemented separately (see Figure 17).
Figure 17: JSR-82 packages

The JSR-82 specification defines three new types of connection (Serial Port Profile, L2CAP, and OBEX), therefore the GCF (java.microedition.io) has to be changed to support these connections. The core Bluetooth API is required in Series 60 Platform JSR-82 implementations, whereas OBEX is not required, thus it is not described in this document.

The core Bluetooth API (javax.bluetooth) is depicted in Figure 18.

Figure 18: Overall architecture and context of core Java Bluetooth API

In Figure 18, the core Bluetooth API is divided into three parts:

1. **Discovery Manager**, which is used by applications for device discovery (inquiry) and service discovery.

2. **Generic Connection Framework (GCF)** is a framework for all connection APIs in J2ME. GCF is defined in the CLDC specification. The Java Bluetooth API defines Bluetooth-specific APIs for the GCF.

3. **Bluetooth Manager** accesses miscellaneous features of the Bluetooth stack. For example, security is handled in Bluetooth Manager.
Typically, a Bluetooth service is an application acting as a server that provides some kind of assistance to client applications via Bluetooth communications. The table below lists some responsibilities of Bluetooth server and client applications.

<table>
<thead>
<tr>
<th>Server</th>
<th>Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creates and adds a service record to the server's Service Discovery Database (SDDB), so it's visible and available to potential clients.</td>
<td>Starts Device discovery to retrieve devices or retrieve an existing list of remote devices.</td>
</tr>
<tr>
<td>Registers the Bluetooth security measures associated with the service (enforced for connections with clients).</td>
<td>Starts a service discovery to retrieve the service records.</td>
</tr>
<tr>
<td>Accepts connections from clients.</td>
<td>Creates a connection URL using the service records.</td>
</tr>
<tr>
<td>Updates (and removes) service records in the SDDB whenever services' attributes change.</td>
<td>Opens a connection to the server.</td>
</tr>
<tr>
<td>Sends data to and receives data from the client.</td>
<td>Sends data to and receives data from the server.</td>
</tr>
</tbody>
</table>

The Java Bluetooth API contains classes LocalDevice and RemoteDevice, which provide device-management capabilities. The LocalDevice class defines the basic functions of the Bluetooth manager (see Figure 18) when the RemoteDevice class represents a remote Bluetooth device. Both classes enable methods to retrieve the device's type and information. The RemoteDevice class also provides methods to authenticate, authorize, or encrypt data transferred between local and remote devices.

The DiscoveryAgent class and DiscoveryListener interface provide methods to find other devices (RemoteDevices) and their services. The DiscoveryAgent object is used to obtain a list of accessible devices and the DiscoveryListener interface is used to get notifications when a new device is found or the inquiry is completed.

```java
//Retrieve the discovery agent
DiscoveryAgent discoveryAgent = local.getDiscoveryAgent();
//Start inquiry
boolean complete = discoveryAgent.startInquiry(int accessCode, DiscoveryListener discoverListener);
```

Because inquiry of remote devices may take some time, the DiscoveryAgent.retrieveDevices method can be used to retrieve an existing list of remote devices if there is no need to find new remote devices. Existing devices may be preknown or cached. Preknown devices are specified by the user as devices with which the local device will frequently communicate. Cached devices are devices that have been found by the local device during previous inquiry requests.

```java
//Retrieving a list of Pre-known and cached devices
```
RemoteDevice[] preKnownDevices =
discoveryAgent.retrieveDevices(DiscoveryAgent.PREKNOWN);

RemoteDevice[] cachedDevices =
discoveryAgent.retrieveDevices(DiscoveryAgent.CACHED);

The client application can search services after a list of remote devices (possible servers) is available. Service discovery is much like device discovery, and DiscoveryAgent also provides methods to discover services on a Bluetooth server device, and to initiate service-discovery transactions. DiscoveryListener provides methods to get notifications when a service is found or the service search is completed or cancelled. In the following code a service search is started:

    //Starting of service search

    int selectServiceTransID =
        discoveryAgent.searchServices(int[] attrSet, UUID[] uuidSet,
                      RemoteDevice remoteDevice, DiscoveryListener discoveryListener);

Before a service can be discovered, it must first be registered and advertised on a Bluetooth server device. To create and register a new service record that represents a service, Connector.open must be invoked and its result must be cast to a StreamConnectionNotifier that represents the service:

    //Open the connection

    StreamConnectionNotifier service = (StreamConnectionNotifier)
    Connector.open("serviceURL");

    //Service is ready to accept client connections. This method blocks
    //until a client connects.

    StreamConnection connection =
        (StreamConnection)service.acceptAndOpen();

    //When exiting the connection is closed and the service record is
    //removed

    service.close();
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 is 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 comprises 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(). There is no other limit for RecordStore or individual records except the amount of free memory. The amount of free memory can be determined by using the RecordStore.getSizeAvailable() method.

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. Trusted and Untrusted MIDlet Suites

Downloaded code can be pretty dangerous, especially when you don’t know who wrote the code or where it came from. Although MIDlets running on a virtual machine are intrinsically a lot safer than binary code, there are still risks.

A downloaded MIDlet may make unauthorized network connections, which can cost a user money. A rogue MIDlet may try to collect information and send it to a server for unauthorized use.

A MIDP device gives its owner a lot of flexibility in choosing games and applications, but with that flexibility comes the risk of running code whose origins and motivations may be dubious. Thus, MIDP 2.0 introduces the concept of Trusted and Untrusted MIDlets.

10.1 Untrusted MIDlet Suites

Untrusted MIDlet suites allow access to nonsensitive APIs as UI components, media control, etc, without explicit confirmation by the user.

Without explicit confirmation by the user, Untrusted MIDlet suites DO NOT allow access to protected APIs like HTTP connection or SMS sending APIs. This provides more security, as it prevents MIDlet to send private data to server.

Confirmation from user permission can be blanket, session, and one shot. Blanket confirmation means that as long as the MIDlet is installed, permission is granted. Session means that as long as the MIDlet is running, permission is granted. One shot means that every time the API is invoked, confirmation is prompted.

10.2 Trusted MIDlet Suites

A signed MIDlet suite assures the user that the contents of the MIDlet suite have not been tampered with and that the MIDlet suite comes from an identifiable (and thus legally accountable) source.

The MIDP implementation on the device can verify the identity of the developer by verifying the developer's signature. It can use the developer's public key to verify the integrity of the MIDlet suite itself.

Security is based on protection domains. Each protection domains defines the permissions that can be granted to the MIDlet suite. The owner of the protection domain specifies how the devices identifies and verifies that it can trust MIDlet suite and get the permission to use protected APIs.

10.3 Protection Domains for Trusted MIDlet Suites

Nokia MIDP 2.0 devices follow the recommended security policy for GMS/UMTS devices as specified in MIDP 2.0 (JSR 118).

The policy recommends four protection domains:
1. Manufacturer domain
2. Operator domain
3. Trusted third-party domain
4. Untrusted domain

The manufacturer's domain will use a root certificate belonging to the device producer. The operator's root certificate will use a root certificate available on storage like the SIM card. The trusted third-party domain will encompass well-known Certificate Authorities (CA) root certificates. The untrusted domain has no root certificate for nonsigned MIDlets or MIDP 1.0 MIDlets.

When creating a MIDlet, that MIDlet needs permission to be added into the .JAD file:

```
MIDlet-Permission: javax.microedition.io.Connector.http
```

The requested permission must be a subset of the permission given to a protection domain, otherwise the MIDlet will not be accepted by the device. If the device can verify the authenticity and integrity of the MIDlet suite, and assign it to a protection domain, the MIDlet suite is said to be trusted. A trusted MIDlet suite will have its requested permissions granted according to its protection domain.

### 10.4 Installation

The process of verifying whether a MIDlet is trusted or not is performed at the time of installation. When the MIDlet is installed, the device will do basic security checks, such as class verification and attribute checks. If the device finds a `MIDlet-Jar-RSA-SHA1` attribute in the .JAD file, it will initiate the authentication and authorization procedures.

During authentication, it will read the chain of certificates in the .JAD file written in the attributes `MIDlet-Certificate<-><m>`, where `n` and `m` are numbers indicating the certificate chain.

If a device can validate the MIDlet certificate with one of the root certificates, it will extract the public key from the certificate and use it to calculate the signature of the .JAR file, comparing it to the protection domain assigned to the root certificate.

If one or more requested permissions on the `MIDlet-permissions` attribute in the .JAD file are not in a protection domain, the installation fails, and is not allowed to continue.
11. Nokia UI API

Nokia UI API is an extension of the standard MIDP 1.0 API and is available in Series 60 devices. Because MIDP was designed to be as portable as possible, certain useful capabilities, such as sound and graphics transparency, were excluded for the reason that they couldn't be implemented by all potential MIDP devices. Nokia's UI API was created to make some of these useful features available, as they are extremely valuable, particularly for game developers.

Many of the features in the Nokia UI API are included in MIDP 2.0 Game API and Media API. However, the Nokia UI API is included, for example, in the Nokia 6600 phone to maintain compatibility.

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 play parameter indicates how many times to repeat the sound.

To start the phone's vibration feature:

```java
DeviceControl.startVibr(100, 500);
```

The parameters to startVibr 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 Web site.

**Note:** Nokia Series 60 devices do not support either vibration or backlight control.

11.1 MIDP 2.0 and Nokia UI API

MIDP 2.0 offers all of the functionalities that the Nokia UI API adds to MIDP 1.0. The Nokia UI API will be deprecated in the future. However, future Nokia devices supporting MIDP 2.0 will also support also the Nokia UI API for purposes of backward compatibility. The following table shows the Nokia UI API and its methods’ counterparts in MIDP 2.0:
<table>
<thead>
<tr>
<th>Functionality</th>
<th>Nokia UI API</th>
<th>MIDP 2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vibration</td>
<td>DeviceControl.startVibra(int strength, int duration)</td>
<td>Display.vibrate(int duration)</td>
</tr>
<tr>
<td>Backlight</td>
<td>DeviceControl.flashLights(int duration)</td>
<td>Display.flashBacklight(int duration)</td>
</tr>
<tr>
<td>Full screen</td>
<td>com.nokia.mid.ui.FullCanvas</td>
<td>Canvas.setFullScreenMode(true)</td>
</tr>
<tr>
<td>Transparency</td>
<td>com.nokia.mid.ui.DirectUtils.createImage(...)</td>
<td>Image.createImage(...)</td>
</tr>
<tr>
<td>Sound support</td>
<td>com.nokia.mid.sound.init(int frequency, int duration)</td>
<td>javax.microedition.media.Manager.playTone(int note, int duration, int volume)</td>
</tr>
</tbody>
</table>
12. An Overview of Mobile Media API (JSR-135)

The JSR-135 specification proposes a Mobile Media API for any J2ME-based virtual machines. This small footprint API allows easy access and control of basic audio and multimedia resources but also addresses scalability and support of more sophisticated features.

MMAPI features the following:

- **Support for Tone Generation, Playback, and Recording of Time-Based Media**: The package supports any time-based audio or video content by offering tools to control the flow of the media stream. Tone generation is a special media type that is characterized by frequency and duration.
- **Small Footprint**: MMAPI works within the strict memory limits of CLDC devices.
- **Protocol and Content Agnostic**: The API is not biased towards any specific content type or protocol.
- **Subsettable**: It is possible to separate a subset of the API in order to provide support for only some type of content (e.g., MIDP 2.0 Media Package)
- **Extensible**: New features can be added easily without breaking older functionality.
- **Options for Implementers**: The API offers features for different purposes. The API is designed to allow implementers to leave some features unimplemented if they cannot be supported.

Furthermore, the JSR-135 specification is closely linked to MIDP 2.0. A subset of JSR-135 has been adopted for MIDP 2.0 Media Package. This enables a seamless manner of application development for mobile devices with different multimedia capabilities.

12.1 Media Processing

Multimedia processing can be broken into two parts:

- **Protocol handling** refers to reading data from a source (such as a file, capture device, or streaming server) into a media processing system.

- **Content handling** usually requires processing the media data (parsing or decoding, for example) and rendering the media to output devices such as an audio speaker or video display.

The API provides two high-level objects – DataSource and Player to encapsulate the two parts of the multimedia processing.

DataSource encapsulates protocol handling. It hides the details of how the data is read from its source, whether the data is coming from a file, streaming server, or proprietary delivery mechanism. DataSource provides a set of methods to allow a Player to read data from it for processing.

A Player reads from the DataSource, processes the data, and renders the media to the output device. It provides a set of methods to control media playback and basic synchronization. Players also provide some type-specific controls to access features for specific media types.

MMAPI also specifies a factory mechanism, the Manager, which creates Players from DataSources. For convenience, the Manager also provides methods to create Players from locators and InputStreams. The overall architecture of MMAPI is shown in Figure 18.
The top-level entry point to the API is through the createPlayer method, which is provided by the Manager object. For example:

```java
Player player = Manager.createPlayer(String url);
```

The `url` fully specifies the protocol and the content of the data, using the format `<protocol>:<content location>`.

The returned `Player` provides general methods to control data flow and presentation. A player’s life cycle includes five states: UNREALIZED, REALIZED, PREFETCHED, STARTED, and CLOSED. Player state transition is shown in Figure 19; six methods are involved.

```
UNREALIZED
  |               close()
  |   deallocate()
  |  realize()
  ↓
REALIZED
  |   prefetch()
  ↓
PREFETCHED
  |  start() or end of media
  ↓
STARTED
  |  close()
  ↓
CLOSED
```

Figure 20: Player state transition

Fine-grained control is an important feature of the API. Therefore, each `Player` also provides type-specific controls with the `getControls` and `getControl` methods:

```java
Control[] Player.getControls()

Control Player.getControl(String controlType)
```
Since each type of media will yield different types of controls from its corresponding Player, the getControls and getControl methods expose features that are unique to a particular media type.

12.2 Developer Platform 2.0 for Series 60 MMAPI Implementation

Developer Platform 2.0 for Series 60 has implemented MMAPI 1.1. The following content types are supported.

- For audio: WAV, MIDI, SP-MIDI, AMR, AMR-WB, MP3, RealAudio, Tone Sequence (audio/x-tong-seq), and Nokia ring tone (audio/x-nokia-rng)
- For video: MPEG-4, H.263, 3GP, and RealVideo

The supported formats of photos shot by the built-in digital camera are PNG and JPEG.

VolumeControl and StopTimeControl are supported by all of the audio contents. VolumeControl, StopTimeControl, and VideoControl are supported by all of the video contents.

MIDIControl, video recording, streaming (audio/video), and mixing are not supported by the implementation. These can be verified by retrieving the corresponding system properties.

As an improvement to Developer Platform 1.0 for Series 60, audio recording is supported.
13. An Overview of Push Registry

MIDP 2.0 includes a new Push Registry feature. It is part of the Application Management System (AMS), and it exposes the Push Registry API, which allows developers to register push alarms and connections and retrieve information about push connections. The AMS can monitor port activity looking for a specific connection type and port, or a MIDlet can be activated at a specific time (see Figure 21). Users can, of course, manually start the MIDlet as before with MIDP 1.0. In the Developer Platform for Series 60, supported inbound connection modes for the Push Registry are SMS and Bluetooth.

Figure 21: MIDlet activation

Registration can take place either during MIDlet suite installation (Figure 242) or dynamically, during some initial execution (Figure 253) of the MIDlet. The former is accomplished using a MIDlet-Push-<n> attribute in the JAD file or in the JAR manifest. The latter is done using the registerConnection() method of the class PushRegistry. The PushRegistry class resides in the javax.microedition.io package. Dynamic and static registrations are presented in Figures 22 and 23. Registration parameters are the same both for static and dynamic registration:

- **ConnectionURL** – The connection string used in Connector.open().
- **MIDletClassName** – The MIDlet that is responsible for the connection. The named MIDlet is registered in the descriptor file or the JAR file manifest with a MIDlet-<n> record. (This information is needed when displaying messages to the user about the application when push connections are detected, or when the user grants/revokes privileges for the application.) If the named MIDlet appears more than once in the suite, the first matching entry is used.
- **AllowedSender** – A designated filter that restricts which senders are valid for launching the requested MIDlet. The syntax and semantics of the AllowedSender field depend on the addressing format used for the protocol. However, every syntax for this field supports using the wildcard characters “*” and “?”. 
A MIDlet can browse through the list of inbound connections of its MIDlet suite with the listConnections() method. This method is also used to get an actual URL for establishing a connection with the originator.

An inbound connection is removed from the PushRegistry in either of the following cases:

A MIDlet suite that has registered the connection is uninstalled;

An application uses the unregisterConnection() method of the PushRegistry class to remove an inbound connection from the list of registered connections.
14. Further Reading


Books
- J2ME in a Nutshell (O'Reilly Java)
- John W. Muchow: Core J2ME Technology