2

Fundamentals of Java ME MIDP Programming

In Chapter 1, we examined the core Mobile Information Device Profile (MIDP) functionality and outlined the CLDC and MIDP classes that form the development environment. In this chapter, we discuss the basic concepts of the MIDP application model, its component pieces and how they fit together to create a fully functional mobile application.

The goal of this chapter is to build a baseline of information to be used in the rest of the book, such as a basic knowledge of application development in MIDP, the MIDP application model, commonly used packages and classes, the packaging and deployment model, and an overview of some the most important optional APIs. The information presented here is also useful for the Symbian OS Java ME certification exam.

We then look at a basic application example which summarizes the process of creating a mobile application from end to end, and how to run it on an emulator and on a real device based on Symbian OS.

If you are an experienced Java ME developer and familiar with MIDP development, you may want to only browse quickly through this chapter, before you move on to Chapter 3, where we explore Java ME on Symbian smartphones specifically.

2.1 Introduction to MIDP

There are three types of component (see Section 1.5) that make up the Java ME environment for mobile devices such as mobile phones: configurations, profiles and optional packages.

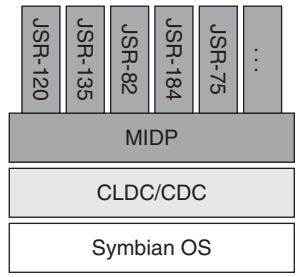

Figure 2.1 Basic Java Micro Edition environment architecture

Figure 2.1 summarizes the architecture of the Java ME environment. Note that Symbian OS is not a mandatory part of the Java ME environment, but we will consider it to be so for the scope of this book.

The MIDP application model defines what a MIDlet is, how it is packaged, and how it should behave with respect to the sometimes constrained resources of an MIDP device. MIDP provides an application framework for mobile devices based on configurations such as CLDC and CDC.

It also defines how multiple MIDlets can be packaged together as a suite, using a single distribution file called a Java Archive (JAR). Each MIDlet suite JAR file must be accompanied by a descriptor file called the JAD file, which allows the Application Management Software (AMS) on the device to identify what it is about to install.

2.2 Using MIDlets

A MIDlet is an application that executes under the MIDP. Unlike a desktop Java application, a MIDlet does not have a `main` method; instead, every such application must extend the `javax.microedition.midlet.MIDlet` class and provide meaningful implementations for its lifecycle methods. MIDlets are controlled and managed by the AMS, part of the device's operating environment. They are initialized by the AMS and then guided by it through the various changes of state of the application. We look briefly at these states next.

2.2.1 MIDlet States

A state is designed to ensure that the behavior of an application is consistent with the expectations of the end users and device manufacturer. Initialization of the application should be short; it should be possible to put an application in a non-active state; and it should also be possible to

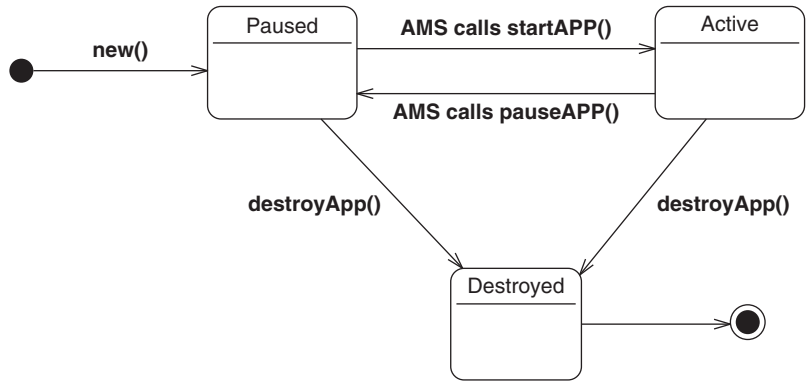

Figure 2.2 MIDlet states

destroy an application at any time. Once a MIDlet has been instantiated, it resides in one of three possible states (see Figure 2.2):

- PAUSED

  The MIDlet has been initialized but is in a dormant state. This state is entered in one of four ways:

  - after the MIDlet has been instantiated; if an exception occurs, the DESTROYED state is entered
  - from the ACTIVE state, if the AMS calls the `pauseApp()` method
  - from the ACTIVE state, if the `startApp()` method has been called but an exception has been thrown
  - from the ACTIVE state, if the `notifyPaused()` method has been invoked and successfully returned.

  During normal execution, a MIDlet may move to the PAUSED state a few times. It happens, for example, if another application is brought to the foreground or an incoming call or SMS message arrives. In these cases, the MIDlet is not active and users do not interact with it. It is, therefore, good practice to release shared resources, such as I/O and network connections, and to stop any running threads and other lengthy operations, so that they do not consume memory, processing resources, and battery power unnecessarily.

- ACTIVE

  The MIDlet is functioning normally. This state is entered after the AMS has called the `startApp()` method. The `startApp()` method can be called on more than one occasion during the MIDlet lifecycle. When a MIDlet is in the PAUSED state, it can request to be moved into the ACTIVE state by calling the `startApp()` method.

- DESTROYED

  The MIDlet has released all resources and terminated. This state, which can only be entered once, can be entered when the `destroyApp(boolean unconditional)` method is called by the AMS and returns successfully. If the `unconditional` argument is false, a `MIDletStateChangedException` may be thrown and the MIDlet will not move to the DESTROYED state. Otherwise, the `destroyApp()` implementation should release all resources and terminate any running threads, so as to guarantee that no resources remain blocked or using memory after the MIDlet has ceased to execute.

  The MIDlet also enters the DESTROYED state when the `notifyDestroyed()` method successfully returns; the application should release all resources and terminate any running threads prior to calling `notify-Destroyed()`.

2.2.2 Developing a MIDlet

Once the source code has been written, we are ready to compile, pre-verify and package the MIDlet into a suite for deployment to a target device or a device emulator.

In this section, we create our first MIDlet in a simple but complete example of MIDlet creation, building, packaging and execution. We use a tool called the Java Wireless Toolkit (WTK) which provides a GUI that wraps the functionality of the command-line tool chain: compiler, pre-verifier and packaging tool.

The WTK (see Figure 2.3) was created by Sun to facilitate MIDP development. It can be obtained free of charge from Sun's website (***java.sun.com/products/sjwtoolkit/download.html***).

The WTK offers the developer support in the following areas:

- **Building and packaging:** You write the source code using your favorite text editor and the WTK takes care of compiling it, pre-verifying the class files, and packaging the resulting MIDlet suite
- **Running and monitoring:** You can directly run applications on the available mobile phone emulators or install them in a process emulating that of a real device. Your MIDlets can be analyzed by the memory monitor, network monitor and method profiler provided by the WTK.
- **MIDlet signing:** A GUI facilitates the process of signing MIDlets, creating new key stores and key pairs, and assigning them to different security domains for testing with the different API permission sets associated with those domains.

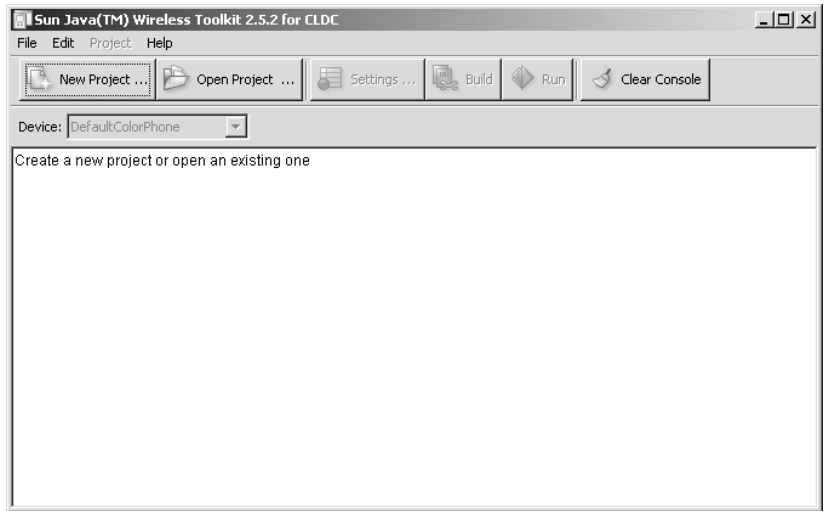

Figure 2.3 Java Wireless Toolkit

- **Example applications:** The Wireless Toolkit comes with several example applications you can use to learn more about programming with MIDP and many optional API packages.

At the time of writing, the WTK is available in production releases for Microsoft Windows XP and Linux-x86 (tested with Ubuntu 6.x). For development, you also require the Java 2 Standard Edition (Java SE) SDK of at least version 1.5.0, available from ***java.sun.com/javase/downloads/index.html***.

Check the WTK documentation, installed by default at `C:\WTK2.5.2\index.html`, for more details on how to use the WTK's facilities for developing MIDlets. Be sure to have installed both the Java SDK and the WTK before trying out our example code.

2.2.3 Creating and Running a MIDlet using the WTK

Now that we are all set with the basic tool, let's create our first MIDlet. The first thing we do is to create a project within the WTK for our new application:

1. Go to Start Menu, Programs, Sun Java Wireless Toolkit for CLDC.
2. Choose Wireless Toolkit and click to start it up.
3. Click on New Project... and enter `Hello World` into the Project Name field and `example.HelloWorldMIDlet` into the MIDlet Class Name field.

4. Click on the Create Project button. In the following screen, change the Target Platform to JTWI and click OK.

Your project is now created, so it's time to write our first MIDlet application. Open a text editor and type in the following code:

```
package example;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Form;
import javax.microedition.midlet.MIDlet;
import javax.microedition.midlet.MIDletStateChangeException;
public class HelloWorldMIDlet extends MIDlet {
  private Form form = null;

  public HelloWorldMIDlet() {
    form = new Form("HelloWorld Form");
    form.append("Hello, World!");
    form.append("This is my first MIDlet!");
  }
  public void destroyApp(boolean arg0) throws MIDletStateChangeException {}
  public void pauseApp() {}
  public void startApp() throws MIDletStateChangeException {
    Display.getDisplay(this).setCurrent(form);
  }
}
```

In the constructor, we create a new `Form` object and append some text strings to it. Don't worry about the details of the `Form` class. For now it's enough to know that `Form` is a UI class that can be used to group small numbers of other UI components. In the `startApp()` method, we use the `Display` class to define the form as the current UI component being displayed on the screen.

Build your MIDlet with the following steps:

1. Save the file, with the name `HelloWorldMIDlet.java`, in the source code folder created by your installation of the Wireless Toolkit. Usually its path is `<home>\j2mewtk\2.5.2\apps\HelloWorld\src\example`, where `<home>` is your home directory. In Windows XP systems, this value can be found in the USERPROFILE environment variable; in Linux, it is the same as the home folder. You must save the `.java` file in the `src\example` folder because this MIDlet is in the `example` package.
2. Switch back to the WTK main window and click Build. The WTK compiles and pre-verifies the `HelloWorldMIDlet` class, which is then ready to be executed.
3. Click on Run. You are presented with a list of MIDlets from the packaged suite, which are ready to be run. As our test suite only has one MIDlet, click on its name and it is executed on the mobile phone emulator (see Figure 2.4).

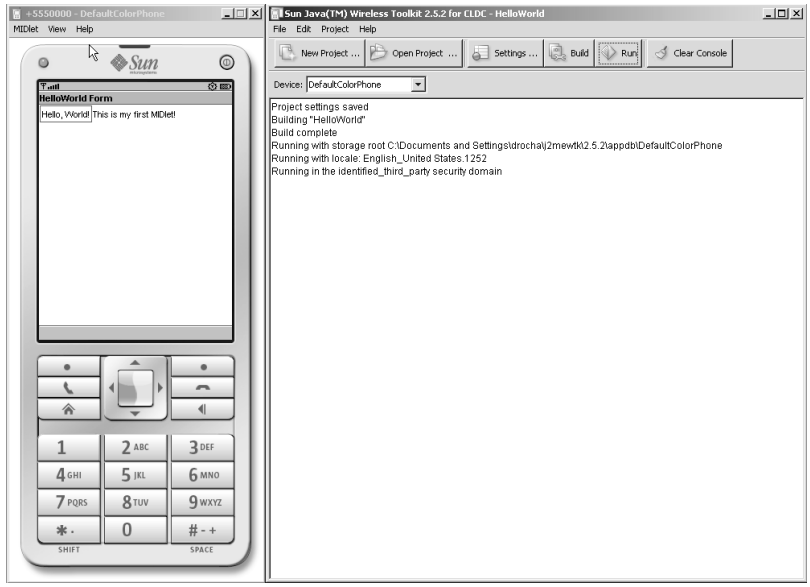

Figure 2.4 Running the `HelloWorld` MIDlet

2.2.4 Packaging a MIDlet

Following the above steps is sufficient to build simple MIDlets for running in the emulator. However, if you are developing a more sophisticated MIDlet that contains many classes, images, application parameters, and so on, you need to package your MIDlet into a MIDlet suite.

Packaging creates a JAR file containing all your class and resource files (such as images and sounds) and the application descriptor (JAD) file, which notifies the AMS of the contents of the JAR file.

The following attributes must be included in a JAD file:

- `MIDlet-Name`: the name of the suite that identifies the MIDlets to the user
- `MIDlet-Version`: the version number of the MIDlet suite; this is used by the AMS to identify whether this version of the MIDlet suite is already installed or whether it is an upgrade, and communicate this information to the user
- `MIDlet-Vendor`: the organization that provides the MIDlet suite
- `MIDlet-Jar-URL`: the URL from which the JAR file can be loaded, as an absolute or relative URL; the context for relative URLs is the place from where the JAD file was loaded
- `MIDlet-Jar-Size`: the number of bytes in the JAR file.

The following attributes are optional but are often useful and should be included:

- `MIDlet-n`: the name, icon and class of the nth MIDlet in the JAR file (separated by commas); the lowest value of n must be 1 and all following values must be consecutive; the name is used to identify the MIDlet to the user and must not be null; the icon refers to a PNG file in the resource directory and may be omitted; the class parameter is the name of the class extending the MIDlet class
- `MIDlet-Description`: a description of the MIDlet suite
- `MicroEdition-Configuration`: the Java ME configuration required, in the same format as the `microedition.configuration` system property, for example, 'CLDC-1.0'
- `MicroEdition-Profile`: the Java ME profiles required, in the same format as the `microedition.profiles` system property, that is 'MIDP-1 0' or 'MIDP-2.0'; if the value of the attribute is 'MIDP-2.0', the target device must implement the MIDP profile otherwise the installation will fail; MIDlets compiled against MIDP 1.0 will install successfully on a device implementing MIDP 2.0.

The following is the JAD file for our HelloWorld application:

```
MIDlet-1: HelloWorldMIDlet,,example.HelloWorldMIDlet
MIDlet-Description: Example MIDP MIDlet
MIDlet-Jar-Size: 1042
MIDlet-Jar-URL: HelloWorld.jar
MIDlet-Name: HelloWorld Midlet Suite
MIDlet-Vendor: Midlet Suite Vendor
MIDlet-Version: 1.0.0
MicroEdition-Configuration: CLDC-1.1
MicroEdition-Profile: MIDP-2.0
```

You can automate the task of creating the JAD file by clicking Package/Create Package in the WTK menu. The WTK creates `HelloWorld.jar` and `HelloWorld.jad` in the `bin` folder of your project root. You can use Windows Explorer to browse to that folder and check that the package files have been correctly created. You can open the JAD file with a standard text editor to inspect its contents. Double-clicking the JAD file executes your application in the emulator (see Figure 2.4).

2.3 MIDP Graphical User Interfaces API

In this section, we take a look at the main APIs provided by the MIDP profile for GUI application development. Designing a user interface

for Java ME applications on mobile devices is quite a challenging task, because MIDP's flexibility allows it to be used in hundreds of different device models, with different form factors, hardware, screen sizes and input methods. Such wide availability makes Java ME applications attractive to a range of users (enterprises, gamers, and casual users), all of whom need to be considered when creating effective UIs for MIDlets.

There are many differences (see Table 2.1) between the hardware and software environments in which Java originated (desktop computers) and the ones found in mobile devices, such as Symbian smartphones.

Table 2.1 Differences between Java environments

Environment	Windows, Linux and Mac OS Personal Computers	Typical Symbian smartphone
Screen	1024 × 768 or bigger Landscape orientation	240 × 320 352 × 416 800 × 352 176 × 208 Landscape or portrait orientation
Input method	QWERTY keyboard Mouse	Numerical keypad QWERTY keyboard Touch screen (with stylus) Touch screen (with fingers)
Display colors	16 bit and higher, usually 32 or 64 bit	16 bit (top phones)
Processing power and memory	1 GHZ+/512 MB RAM +	Around 350 MHz

These differences (and the many others that exist) make it inappropriate to port Swing or AWT toolkits directly to mobile devices. They would suffer from poor performance (due to slower processors and smaller memory), usability problems (there cannot be multiple windows on a mobile phone) and poor input mechanism compatibility, as this varies greatly among devices.

2.3.1 LCDUI Model for User Interfaces

The LCDUI toolkit is a set of features for the implementation of user interfaces especially for MIDP-based devices. LCDUI is a generic set of UI components split into two categories, high-level and low-level.

The high-level UI components' main characteristics are a consistent API and leaving the actual UI layout and appearance to be implemented by the device environment itself. Developers can write their code against a single set of classes and objects and trust they will appear consistently in all devices implementing that set. This approach ensures that high-level UI components are portable across devices, appearing on the screen in a manner that is consistent with the device's form factor, screen size and input methods. The downside is that little control is left in the developers' hands, and some important things, such as fonts, colors and component positioning, can only be hinted at by the developer – the implementation is free to follow those hints or not.

The low-level set of UI components consists of the `Canvas` class, the `GameCanvas` subclass and the associated `Graphics` class. They provide fine-grained, pixel-by-pixel control of layout, colors, component placement, etc. The downside of using the low-level set is that the developer must implement basic UI controls (dialogs, text input fields, forms), since the `Canvas` is simply a blank canvas which can be drawn upon. The `CustomItem` class of the `javax.microedition.lcdui` package can be thought of as belonging to the low-level UI set, as it provides only basic drawing functionalities, allowing developers to specify layout and positioning of controls very precisely. However, as custom items are only used within `Forms` they are discussed in Section 2.3.2.

The Event Model

The `javax.microedition.lcdui` package implements an event model that runs across both the high- and low-level APIs. It handles such things as user interaction and calls to redraw the display. The implementation is notified of such an event and responds by making a corresponding call back to the MIDlet. There are four types of UI event:

- events that represent abstract commands that are part of the high-level API; the Back, Select, Exit, Cancel commands seen in Symbian OS devices generally fit this category
- low-level events that represent single key presses or releases or pointer events
- calls to the `paint()` method of the `Canvas` class
- calls to an object's `run()` method.

Callbacks are serialized and never occur in parallel. More specifically, a new callback never starts while another is running; this is true even when there is a series of events to be processed. In this case, the callbacks are processed as soon as possible after the last UI callback has returned. The implementation also guarantees that a call to `run()`, requested by

a call to `callSerially()`, is made after any pending `repaint()` requests have been satisfied. There is, however, one exception to this rule: when the `Canvas.serviceRepaints()` method is called by the MIDlet, it causes the `Canvas.paint()` method to be invoked by the implementation and then waits for it to complete. This occurs whenever the `serviceRepaints()` method is called, regardless of where the method was called from, even if that source was an event callback itself.

The `Command` Class

Abstract commands are used to avoid having to implement concrete command buttons; semantic representations are used instead. The commands are attached to displayable objects, such as high-level `List` or `Form` objects or low-level `Canvas` objects. The `addCommand()` method attaches a command to the displayable object. The command specifies the label, type and priority. The `CommandListener` interface then implements the actual semantics of the command. The native style of the device may prioritize where certain commands appear on the UI. For example, Exit is always placed above the right softkey on Nokia devices. There are also some device-provided operations that help contribute towards the operation of the high-level API. For example, screen objects, such as `List` and `ChoiceGroup`, have built-in events that return user input to the application for processing.

2.3.2 LCDUI High-Level API: Screen Objects

`Alert`, `List`, `TextBox`, and `Form` objects are all derived from `Screen`, itself derived from `Displayable`. `Screen` objects are high-level UI components that can be displayed. They provide a complete user interface, of which the specific look and feel is determined by the implementation. Only one `Screen`-derived object can be displayed at a time. Developers can control which `Screen` is displayed by using the `setCurrent()` method of the `Display` class.

This section describes the high-level API classes in a succinct manner, rather than going into every detail. To find a complete description of each featured class, please check the MIDP documentation.

`Alert` Object

An `Alert` object shows a message to the user, waits for a certain period and then disappears, at which point the next displayable object is shown. An `Alert` object is a way of informing the user of any errors or exceptional conditions. It may be used by the developer to inform the user that:

- an error or other condition has been reached

- a user action has been completed successfully
- an event for which the user has previously requested notification has been completed.

To emphasize these states to the user, the `AlertType` can be set to convey the context or importance of the message. For each use case described above, there's a relevant type, such as ALARM, CONFIRMATION, ERROR and INFO. These are differentiated by titles at the top of the alert screen, icons drawn on the alert, and the sound played when it is displayed.

`List` Object

A `List` object is a screen object that contains a list of choices for the user and is, therefore, ideal for implementing choice-based menus, which are the core user interface of most mobile devices.

`TextBox` Object

A `TextBox` is a `Screen` object that allows the user to enter and edit text in a separate space away from the form. It is a `Displayable` object and can be displayed on the screen in its own right. Its maximum size can be set at creation, but the number of characters displayed at any one time is unrelated to this size and is determined by the device itself.

`Form` Object

A `Form` object is designed to contain a small number of closely-related user interface elements. Those elements are, in general, subclasses of the `Item` class and we shall investigate them in more detail below. The `Form` object manages the traversal, scrolling and layout of the items.

Items enclosed within a form may be edited using the `append()`, `delete()`, `insert()` and `set()` methods. They are referred to by their indexes, starting at zero and ending with `size()-1`. Items are organized via a layout policy that is based around rows. The rows typically relate to the width of the screen and are constant throughout. Forms grow vertically and a scroll bar is introduced as required. If a form becomes too large, it may be better for the developer to create another screen. Users can interact with a `Form` and a `CommandListener` can be attached to capture this input using the `setCommandListener()` method. An individual `Item` can be given an `ItemCommandListener`, if a more contextual approach is required by the UI.

`Item` Class

This is the superclass for all items that can be added to a `Form`. Every `Item` has a label, which is a string. This label is displayed by the implementation as near as possible to the `Item`, either on the same horizontal row or above. When an `Item` is created, by default it is not owned by any container and does not have a `Command` or `ItemCommandListener`. However, default commands can be attached to an `Item`, using the `setDefaultCommand()` method, which makes the user interface more intuitive for the user. A user can then use a standard gesture, such as pressing a dedicated selection key or tapping on the item with a pointer. Symbian devices support these interfaces through S60 and UIQ, respectively.

The following types are derived from `Item`.

`ChoiceGroup`

A group of selectable objects may be used to capture single or multiple choices in a `Form`. It is a subclass of `Item` and most of its methods are implemented via the `Choice` interface.

A `ChoiceGroup` has similar features to a `List` object, but it's meant to be placed in a `Form`, not used as a standalone `Screen` object. Its type can be either EXCLUSIVE (to capture one option from the group) or MULTIPLE (to capture many selections). As usual with high-level UI components, developers don't have control over the graphical representation of a `ChoiceGroup`. Usually, though, one of EXCLUSIVE type is shown as a list of radio buttons, while the MULTIPLE type is rendered as a list of checkboxes.

`CustomItem`

`CustomItem` operates in a similar way to `Canvas`: the developer can specify precisely what content appears where within its area. Some of the standard items may not give quite the required functionality, so it may be better to define home-made ones instead. The drawback to this approach is that, as well as having to draw all the contents using the item's `paint()` method, the developer has to process and manage all events, such as user input, through `keyPressed()`. Custom items may interact with either keypad- or pointer-based devices. Both are optional within the specification and the underlying implementation will signal to the item which has been implemented.

`CustomItem` also inherits from `Item`, therefore inheriting the `getMinContentWidth()` and `getPrefContentHeight()` methods, which help the implementation to determine the best fit of items within

the screen layout. If the `CustomItem` is too large for the screen dimensions, it resizes itself to within those preferred, minimum dimensions and the `CustomItem` is notified via `sizeChanged()` and `paint()` methods.

Additionally, the developer can use the `Display.getColor(int)` and `Font.getFont(int)` methods to determine the underlying properties for items already displayed in the form of which the `CustomItem` is a part, to ensure that a consistent appearance is maintained.

DateField

This is an editable component that may be placed upon a `Form` to capture and display date and time (calendar) values. The item can be added to the form with or without an initial value. If the value is not set, a call to the `getDate()` method returns `NULL`. The field can handle DATE values, TIME values, and DATE_TIME values.

ImageItem

The `ImageItem` is a reference to a mutable or immutable image that can be displayed on a `Form`. We look at the `Image` object in detail in Section 2.3.4. Suffice to say that the `Image` is retrieved from the MIDlet suite's JAR file in order to be displayed upon the form. This is performed by calling the following method, in this case from the root directory:

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

An `ImageItem` can be rendered in various modes, such as PLAIN, HYPERLINK, or BUTTON. Please check the MIDP documentation for more details.

Gauge

This component provides a visual representation of an integer value, usually formatted as a bar graph. Gauges are used either for specifying a value between zero and a maximum value, or as a progress or activity monitor.

Spacer

This blank, non-interactive item with a definable minimum size is used for allocating flexible amounts of space between items on a form and gives the developer much more control over the appearance of a form. The minimum width and height for each spacer can be defined to provide space between items within a row or between rows of items on the form.

StringItem

This is a display-only item that can contain a string and the user cannot edit the contents. Both the label and content of the `StringItem` can, however, be changed by the application. As with `ImageItem`, its appearance can be specified at creation as one of PLAIN, HYPERLINK or BUTTON. The developer is able to set the text, using the `setText()` method, and its appearance, using `setFont()`.

TextField

A `TextField` is an editable text component that may be placed in a `Form`. It can be given an initial piece of text to display. It has a maximum size, set by `setSize(int size)`, and an input mask, which can be set when the item is constructed. An input mask is used to ensure that end users enter the correct data, which can reduce user frustration. The following masks can be used: ANY, EMAILADDR, NUMERIC, PHONENUMBER, URL, and DECIMAL. These constraints can be set using the `setConstraints()` method and retrieved using `getConstraints()`. The constraint settings should be used in conjunction with the following set of modifier flags using the bit-wise AND (`&`) operator: PASSWORD, SENSITIVE, UNEDITABLE, NON_PREDICTIVE, INITIAL_CAPS_WORD, INITIAL_CAPS_SENTENCE.

Ticker

This implements a ticker-tape object – a piece of text that runs continuously across the display. The direction and speed of the text is determined by the device. The ticker scrolls continuously and there is no interface to stop and start it. The implementation may pause it when there has been a period of inactivity on the device, in which case the ticker resumes when the user recommences interaction with the device.

2.3.3 LCDUI Interfaces

The user interface package, `javax.microedition.lcdui`, provides four interfaces that are available to both the high- and low-level APIs:

- `Choice` defines an API for user-interface components, such as `List` and `ChoiceGroup`. The contents of these components are represented by strings and images which provide a defined number of choices for the user. The user's input can be one or more choices and they are returned to the application upon selection.
- `CommandListener` is used by applications that need to receive high-level events from the implementation; the listener is attached to

a displayable object within the application using the `addCommand()` method.

- `ItemCommandListener` is a listener type for receiving notification of commands that have been invoked on `Item` objects. This provides the mechanism for associating commands with specific `Form` items, thus contextualizing user input and actions according to the current active item on the form, making it more intuitive.
- `ItemStateListener` is used by applications that need to receive events that indicate changes in the internal state of the interactive items within a form; for example, a notification is sent to the application when the set of selected values within a `ChoiceGroup` changes.

2.3.4 LCDUI Low-Level API: `Canvas`

The low-level API allows developers to have total control of how the user interface looks and how components are rendered on the screen. `Canvas`, the main base class for low-level UI programming, is used to exercise such fine-grained control. An application should subclass `Canvas` to create a new displayable screen object. As it is displayable, it can be used as the current display for an application just like the high-level components. Therefore a MIDlet application can have a user interface with, for example, `List`, `Form` and `Canvas` objects, which can be displayed one at a time to provide the application functionality to the users.

`Canvas` is commonly used by game developers when creating sprite animation and it also forms the basis of `GameCanvas`, which is part of the Game API (see Section 2.3.6). `Canvas` can be used in normal mode, which allows title and softkey labels to be displayed, and full-screen mode, where `Canvas` takes up as much of the display as the implementation allows. In either mode, the dimensions of the `Canvas` can be accessed using the `getWidth()` and `getHeight()` methods.

Graphics are drawn to the screen by implementing code in the abstract `paint()` method. This method must be present in the subclass and is called as part of the event model. The event model provides a series of user-input methods such as `keyPressed()` and `pointerPressed()`, depending upon the device's data input implementation. The `paint(Graphics g)` method passes in a `Graphics` object, which is used to draw to the display.

The `Graphics` object provides a simple 2D, geometric-rendering capability, which can be used to draw strings, characters, images, shapes, etc. For more details, please check the MIDP documentation.

Such methods as `keyPressed()` and `pointerPressed()` represent the interface methods for the `CommandListener`. When a key is pressed, it returns a key code to the command listener. These key codes are

mapped to keys on the keypad. The key code values are unique for each hardware key, unless keys are obvious synonyms for one another. These codes are equal to the Unicode encoding for the character representing the key. Examples of these are KEY_NUM0, KEY_NUM1, KEY_STAR, and KEY_POUND. The problem with these key codes is that they are not necessarily portable across devices: other keys may be present on the keypad and may form a distinct list from those described previously. It is therefore better, and more portable, to use game actions instead. Each key code can be mapped to a game action using the `getGameAction(int keyCode)` method. This translates the key code into constants such as LEFT, RIGHT, FIRE, GAME_A and GAME_B. Codes can be translated back to key codes by using `getKeyCode(int gameAction)`. Apart from making the application portable across devices, these game actions are mapped in such a way as to suit gamers. For example, the LEFT, RIGHT, UP and DOWN game actions might be mapped to the 4, 6, 2 and 8 keys on the keypad, making game-play instantly intuitive.

A simple `Canvas` class might look like this:

```
import javax.microedition.lcdui.*;
  public class SimpleCanvas extends Canvas {
    public void paint(Graphics g) {
      // set color context to be black
      g.setColor(255, 255, 255);
      // draw a black filled rectangle
      g.fillRect(0, 0, getWidth(), getHeight());
      // set color context to be white
      g.setColor(0, 0, 0);
      // draw a string in the top left corner of the display
      g.drawString("This is some white text", 0, 0, g.TOP | g.LEFT);
    }
  }
```

2.3.5 Putting It All Together: `UIExampleMIDlet`

We use our new knowledge of the high- and low-level APIs to build a showcase of most LCDUI components: `Form`, `Item` (`ImageItem`, `StringItem`, `ChoiceGroup`, `DateField` and `Gauge`), `Ticker`, `List`, `TextBox`, and `Canvas`. We also use the `Command` class and its notification interface, `CommandListener`, to show how you can handle these abstract events to produce concrete behavior in your application.

The application also shows how to build a menu using `Command`s and how to create high- and low-level UI components and switch between them, emulating what we would have in a real-world application. For the sake of simplicity and readability, we skip some code sections which are common knowledge among Java programmers, such as the list of imported packages and empty method implementations. This allows us

to focus instead on the behavior of the UI components we have described in so much detail.

To get started, open the WTK and create a new project, called `UIExampleMIDlet`. Our application has two classes in the `example` package: `UIExampleMIDlet` (the main class) and `MenuCanvas`.

`UIExampleMIDlet` Class

```
Class UIExampleMIDlet
package example;
/*
Import list omitted for readability
*/
public class UIExampleMIDlet extends MIDlet implements CommandListener {
  private Form form = null;
  private List list = null;
  private TextBox textBox = null;
  private Canvas canvas = null;
  private TextField textField = null;
  private DateField dateField = null;
  private Alert alert = null;
  private ChoiceGroup choiceGroup = null;
  private StringItem stringItem = null;
  private Image image = null;
  private ImageItem imageItem = null;
  private Command backCommand = null;
  private Command exitCommand = null;
  private Ticker ticker = null;
  private Gauge gauge = null;

  public UIExampleMIDlet() throws IOException {
    // creating commands
    backCommand = new Command("Back",Command.BACK,0);
    exitCommand = new Command("Exit",Command.EXIT,0);

    // creating Form and its items
    form = new Form("My Form");
    textField = new TextField("MyTextField","",20,TextField.ANY);
    dateField = new DateField("MyDateField",DateField.DATE_TIME);
          stringItem = new StringItem("MyStringItem","My value");
    choiceGroup = new ChoiceGroup("MyChoiceGroup",Choice.MULTIPLE,
              new String[] {"Value 1","Value 2","Value 3"}, null);
    image = Image.createImage("/test.png");
    imageItem = new ImageItem("MyImage",image,ImageItem.LAYOUT_CENTER,
                                                        "No Images");
    gauge = new Gauge("My Gauge",true,100,1);

    form.append(textField);
    form.append(dateField);
    form.append(stringItem);
    form.append(imageItem);
    form.append(choiceGroup);
    form.append(gauge);
    form.addCommand(backCommand);
    form.setCommandListener(this);
```

```
  // creating List
  list = new List("MyList",Choice.IMPLICIT,
                              new String[] {"List item 1","List item 2",
                                                  "List item 3"},null);
  list.addCommand(backCommand);
  list.setCommandListener(this);

  // creating textBox
  textBox = new TextBox("MyTextBox","",256,TextField.ANY);
  textBox.addCommand(backCommand);
  textBox.setCommandListener(this);

  // creating main canvas
  canvas = new MenuCanvas();
  canvas.addCommand(exitCommand);
  canvas.setCommandListener(this);
  // creating ticket
  ticker = new Ticker("My ticker is running....");
  canvas.setTicker(ticker);

  // creating alert
  alert = new Alert("Message","This is a message!",null,AlertType.INFO);
}

public void commandAction(Command c, Displayable d) {
  if(c == exitCommand) {
    notifyDestroyed();
  }

  if(c == backCommand) {
    setDisplayable(canvas);
  }

  if(d == canvas) {
    if(c.getLabel().equals("Form")) {
      setDisplayable(form);
    }
    else if(c.getLabel().equals("List")) {
      setDisplayable(list);
    }
    else if(c.getLabel().equals("TextBox")) {
      setDisplayable(textBox);
    }
    else if(c.getLabel().equals("Canvas")) {
      setDisplayable(canvas);
    }
    else if(c.getLabel().equals("Alert")) {
      setDisplayable(alert);
    }
  }
}

private void setDisplayable(Displayable d) {
  Display.getDisplay(this).setCurrent(d);
}
protected void startApp() throws MIDletStateChangeException {
  setDisplayable(canvas);
}
```

```
  protected void destroyApp(boolean unconditional)
              throws MIDletStateChangeException {
    // empty implementation, as there are no resources to be released
  }
  protected void pauseApp() {
    // empty implementation, as there are no resources to be released
  }
}
```

All of our instance variables, references to the UI components in our application, are declared and initialized to `NULL`. In the class constructor, we assign each variable to a newly-created object of the proper class. First, we create a couple of commands (Back and Exit) that will be used to handle the high-level user actions defined by the device implementation, assigned to the UI components and passed to us via the `commandAction(Command, Displayable)` method:

```
backCommand = new Command("Back",Command.BACK,0);
exitCommand = new Command("Exit",Command.EXIT,0);
```

Following this, we begin to create the actual UI components. For example, let's see how a new `Form` object is constructed:

```
form = new Form("My Form");
textField = new TextField("MyTextField","",20,TextField.ANY);
dateField = new DateField("MyDateField",DateField.DATE_TIME);
stringItem = new StringItem("MyStringItem","My value");
choiceGroup = new ChoiceGroup("MyChoiceGroup",Choice.MULTIPLE,
           new String[] {"Value 1","Value 2","Value 3"}, null);
image = Image.createImage("/test.png");
imageItem = new ImageItem("MyImage",image,ImageItem.LAYOUT_CENTER,
                                                    "No Images");
gauge = new Gauge("My Gauge",true,100,1);

form.append(textField);
form.append(dateField);
form.append(stringItem);
form.append(imageItem);
form.append(choiceGroup);
form.append(gauge);
form.addCommand(backCommand);
form.setCommandListener(this);
```

As outlined before, a `Form` is a flexible UI component that can hold other components, instances of the `Item` class, and show them on the screen. Users expect `Form` components to present information that is related in meaning, such as settings for an application. Doing otherwise can be confusing and result in bad user interaction with the application.

After instantiating a `Form`, we create its items (`textField`, `dateField`, `stringItem`, `choiceGroup`, `imageItem` and `Gauge`) and use

the `Form.append()` method to add them to the `Form` components list. They are then displayed with the `Form`. One important detail: the `Image` object we use to create the `ImageItem` retrieves a PNG image from the root of the JAR file. In order to replicate this behavior in your own project, just copy a file called `test.png` to the `res` folder of the project.

After creating and appending the items, we add the Back command to the `Form`; the emulator's implementation maps it to one of the softkeys. Finally, we set the `CommandListener` to be our own MIDlet, which implements the `commandAction()` method (defined in the `CommandListener` interface) to handle user actions.

The rest of the constructor creates the other UI components, adds the Back command to them and sets the MIDlet itself as their event listener. The following lines create an instance of `MenuCanvas` and add the Exit command to it:

```
canvas = new MenuCanvas();
canvas.addCommand(exitCommand);
canvas.setCommandListener(this);
```

This canvas is the main screen for our application.

The `commandAction()` method reacts to the events of the Back and Exit buttons and the `MenuCanvas`, changing screens according to the user's choices. The `startApp()` method simply sets the `MenuCanvas` instance as the main display, using the `setDisplayable()` utility.

MenuCanvas Class

```
package example;
import javax.microedition.lcdui.Canvas;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.Graphics;
public class MenuCanvas extends Canvas {
  private Command[] options = new Command[] {
    new Command("Form",Command.OK,1),
    new Command("List",Command.OK,2),
    new Command("Canvas",Command.OK,3),
    new Command("TextBox",Command.OK,4),
    new Command("Alert",Command.OK,4)
  };

  public MenuCanvas() {
    for(int i=0;i<options.length;i++) {
      this.addCommand(options[i]);
    }
  }

  protected void paint(Graphics g) {
    g.setColor(0,0,0);
    g.fillRect(0,0,getWidth(),getHeight());
```

```
    g.setColor(255,255,255);
    g.drawString("This is a canvas.",0,0,Graphics.TOP | Graphics.LEFT);
    g.drawString("Check Options menu",0,20, Graphics.TOP | Graphics.LEFT);
    g.drawString("for more UI components",0,40,
                 Graphics.TOP | Graphics.LEFT);
  }
}
```

The `options` instance variable holds an array of `Command`s which serves as the main menu of the application. Each command has a label, a positioning guide (`Command.OK`) and a priority value. In the constructor, we loop through the array, adding all the commands to the `Canvas` so they are displayed on the screen. The abstract `paint(Graphics)` method is implemented so that the `Canvas` can be drawn upon, using the `Graphics` object passed in as a parameter. In this case, we set the context color to black, paint a screen-sized rectangle, set the color to white and draw some strings to instruct the user how to use the application.

Save `UIExampleMIDlet` in a file called `UIExampleMIDlet.java` and `MenuCanvas` in a file called `MenuCanvas.java` in your project's `src/example` folder, then build and package the project using the WTK. Figure 2.5 shows the application running on the WTK emulator.

Figure 2.5 `UIExampleMIDlet` application running on the WTK emulator: a) the `MenuCanvas` and b) the `Form`

2.3.6 Game API

The MIDP specification supports easy game development through the use of the `javax.microedition.lcdui.game` package. It contains the following classes:

- `GameCanvas`
- `LayerManager`
- `Layer`
- `Sprite`
- `TiledLayer`.

The aim of the API is to facilitate richer gaming content through a set of APIs that provides useful functionality.

GameCanvas

A basic game user interface class that extends `javax.microedition.lcdui.Canvas`, `GameCanvas` provides an offscreen buffer as part of the implementation even if the underlying device doesn't support double buffering. The `Graphics` object obtained from the `getGraphics()` method is used to draw to the screen buffer. The contents of the screen buffer can then be rendered to the display synchronously by calling the `flushGraphics()` method. The `GameCanvas` class also provides the ability to query key states and return an integer value in which each bit represents the state of a specific key on the device:

```
public int getKeyStates();
```

If the bit representing a key is set to 1, then this key has been pressed at least once since the last invocation of the method. The returned integer can be ANDed against a set of predefined constants, each representing a specific key, by having the appropriate bit set (support for the last four values is optional).

We use the `GameCanvas` class members that describe key presses (e.g., `GameCanvas.FIRE_PRESSED`) to ascertain the state of a key in the manner shown below:

```
if ( getKeyStates() & Game.Canvas.FIRE_PRESSED != 0 ) {
  // FIRE key is down or has been pressed - take appropriate action
}
```

TiledLayer

The abstract `Layer` class is the parent class of `TiledLayer` and `Sprite`. A `TiledLayer` consists of a grid of cells each of which can be filled with an image tile. An instance of `TiledLayer` is created by invoking the constructor:

```
public TiledLayer(int columns, int rows, Image image, int tileWidth,
                                                      int tileHeight);
```

The `columns` and `rows` arguments represent the number of columns and rows in the grid. The `tileWidth` and `tileHeight` arguments represent the width and height of a single tile in pixels. The `image` argument represents the image used to create the set of tiles that populate the `TiledLayer`. Naturally, the dimension of the image in pixels must be an integral multiple of the dimension of an individual tile. The use of `TiledLayer` is best illustrated with an example.

One of the principal uses of `TiledLayer` is the creation of large scrolling backgrounds from relatively few tiles. Consider the example in Figure 2.6, which shows a big set of tiles of equal dimensions that we can arrange in different ways to create a `TiledLayer` background using selected tiles. You can for example reuse tile 1 to create a large lake or tiles 1, 5 and 16 to create a strip of land surrounded by a small

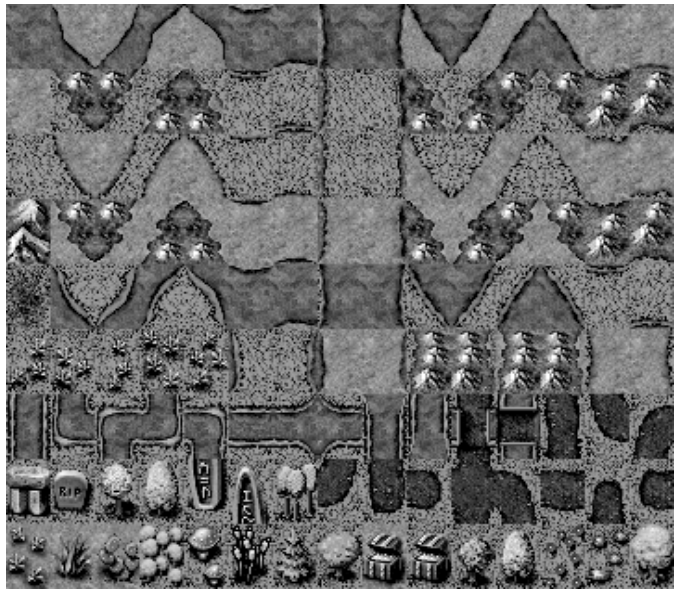

Figure 2.6 Image for use by a `TiledLayer`

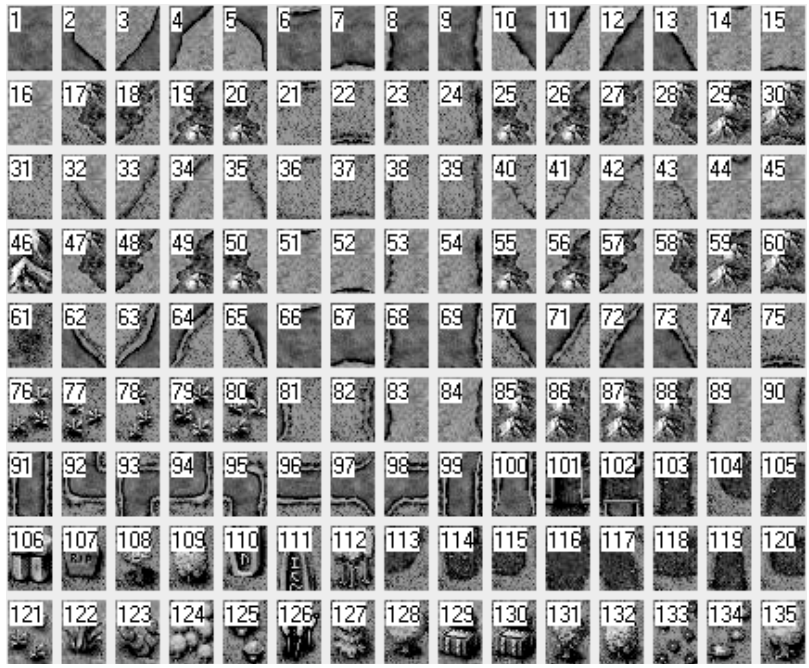

Figure 2.7 Image split into 135 tiles

lake. The dimensions of the image are 360 × 315 pixels and each tile's dimension is 24 × 35 pixels; that gives us a total of 135 tiles which can be treated as a 15 × 9 array (see Figure 2.7) and manipulated to create our background.

We put `TiledLayer` scrolling backgrounds into use later on, with a `LayerManagerDemo` example.

Sprite

A `Sprite` is a basic visual element suitable for creating animations and consists of an image composed of several smaller images (frames). The `Sprite` can be rendered as one of the frames. By rendering different frames in a sequence, a `Sprite` provides animation. Let us consider a simple example. Figure 2.8 consists of 12 frames of the same width and height. By displaying some of the frames in a sequence, we can produce an animation.

Here's how to create and animate a sprite based on the image in Figure 2.8. The code is abbreviated for clarity and we will see a more complete example in the `LayerManagerDemo` MIDlet:

```
// create image for sprite
Image image = Image.createImage("/example_sprite.png");
// create and position sprite
guy = new Sprite(image, SPRITE_WIDTH, SPRITE_HEIGHT);
guy.setPosition(spritePositionX, spritePositionY);
// paint sprite on the screen
public void paint(Graphics g) {
  g.setColor(255, 255, 255);
  g.fillRect(0, 0, getWidth(), getHeight());
  guy.paint(g);
}
// loop through all the frames
while (running) {
  repaint();
  guy.nextFrame();
  ...
}
```

Figure 2.8 A `Sprite` image consisting of 12 frames

In addition to various transformations such as rotation and mirroring, the `Sprite` class also provides collision detection, which is essential for games as it allows the developer to detect when the sprite collides with another element and prevent the hero from crossing a solid wall or obstacle. Both pixel-level and bounding-rectangle collisions are available.

LayerManager

As the name implies, the `LayerManager` manages a series of `Layer` objects. `Sprite` and `TiledLayer` both extend `Layer`. More specifically, a `LayerManager` controls the rendering of `Layer` objects. It maintains an ordered list so that they are rendered according to their z-values (in standard computer graphics terminology). We add a `Layer` to the list using the method:

```
public void append(Layer l);
```

The first layer appended has index zero and the lowest z-value – that is, it appears closest to the user (viewer). Subsequent layers have successively

greater z-values and indices. Alternatively, we can add a layer at a specific index using the method:

```
public void insert(Layer l, int index);
```

To remove a layer from the list we use the method:

```
public void remove(Layer l);
```

We position a layer in the `LayerManager`'s coordinate system using the `setPosition()` method. The contents of `LayerManager` are not rendered in their entirety; instead, a view window is rendered using the `paint()` method of the `LayerManager`:

```
public void paint(Graphics g, int x, int y);
```

The `x` and `y` arguments are used to position the view window on the displayable object (`Canvas` or `GameCanvas`) upon which the `LayerManager` is rendered. The size of the view window is set using this method:

```
public void setViewWindow(int x, int y, int width, int height);
```

The `x` and `y` values determine the position of the top left corner of the rectangular view window in the coordinate system of the `LayerManager`. The `width` and `height` arguments determine the width and height of the view window and are usually set to a size appropriate for the device's screen. By varying the x and y coordinates we can pan through the contents of the `LayerManager`.

`LayerManagerDemo` Example

Our `LayerManagerDemo` example summarizes all Game API concepts seen so far. It illustrates the use of a `Sprite` (for animating the hero), a `TiledLayer` (for background construction), a `LayerManager` (for scrolling the background), and a `GameCanvas` (for drawing it all). The source code is too large to be included in this book, so it is available for download from the website. We highlight here some parts that show how to use the Game API components.

In the constructor, we load the image resources used by our `TiledLayer` and the `Sprite` (see Figures 2.6 and 2.8, respectively). We create our game hero, using the `Sprite` constructor, then set the frame sequence, which allows us to loop only to the frames that interest us. In

this case, we want the hero to walk downwards, so we choose to use only frames 6, 7 and 8:

```
guy = new Sprite(guyImage,SPRITE_WIDTH,SPRITE_HEIGHT);
guy.setFrameSequence(new int[] {6,7,8});
```

We then create the background for the game scene, by constructing a new `TiledLayer` from the source image (see Figure 2.9), and use the `fillLayer()` method and the cells array to create a terrain over which our hero will walk.

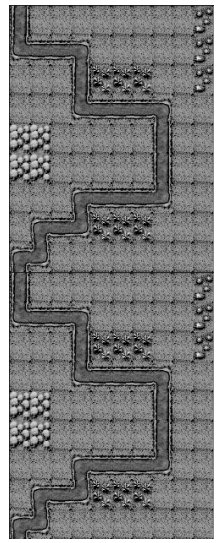

Figure 2.9 Background image for the `LayerManagerDemo` MIDlet

We create a `LayerManager` to manage both the background `TiledLayer` and the `Sprite` (both inherit from `Layer`). We append the `Sprite` then the background layer, to ensure that the hero is shown over the background and not the other way around, since the hero has a smaller z-index. This is how it's done in code:

```
// creating LayerManager
manager = new LayerManager();
manager.append(guy);
manager.append(background);
manager.setViewWindow(0,0,SCREEN_WIDTH,SCREEN_HEIGHT);
```

We also set the view window to match the size of the screen so we can see it in its entirety, and not just a portion of it. Try playing with the values for the view window and you get partial views of the background

that you can use for other games ideas. The last line of code in the constructor just sets the `GameCanvas` to be in full-screen mode.

Since your animation runs permanently, it's good practice to put it in another thread, so we don't block the current thread, which serves UI application requests. This frees the application to continue processing events. To do this, we make our `GameCanvas` `Runnable`, implement the game animation loop within the `run()` method, and use the `start()` and `stop()` methods to start and stop the animation:

```
public void start() {
  running = true;
  Thread t = new Thread(this);
  t.start();
}

public void stop() {
  running = false;
}
```

Using `t.start()` triggers the `run()` method, which is then responsible for the game animation. This includes managing background scrolling, animating the sprite, drawing everything to the screen, and resetting the background scrolling when needed, so our hero won't fall off the screen:

```
public void run() {
  Graphics graphics = this.getGraphics();
  // graphics.setColor(255, 255, 255);

  while(running) {
    layerY -= 1;
    background.setPosition(0,layerY);
    guy.setPosition(getWidth()/2 - guy.getWidth()/2,
                getHeight()/2 - guy.getHeight()/2);
    manager.paint(graphics,0,0);
    flushGraphics();
    guy.nextFrame();

    try {
      Thread.sleep(20);
    }
    catch(InterruptedException e) {
      break;
    }

    if(layerY == -(background.getHeight() / 2)) {
      layerY = 0;
    }
  }
}
```

The source code, JAR and JAD files for the `LayerManagerDemo` MIDlet are available from the book's website.[1] I strongly encourage you

[1] ***developer.symbian.com/javameonsymbianos***

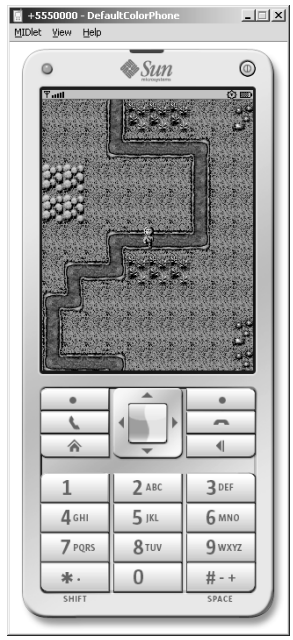

Figure 2.10 `LayerManagerDemo` MIDlet

to at least run the example application with the WTK emulator (see Figure 2.10) so you can see for yourself how these techniques combine to generate the illusion of motion that is the basis of all games and get motivated to create your own games using this skeleton code.

This concludes our introduction to the Game API of MIDP 2.0. We come back to this subject in much more detail in Chapter 8, where you develop a full, multimedia-rich game application expanded to allow multiple player, interconnected gaming.

2.4 Non-GUI APIs in MIDP

In this section, we cover in more detail the MIDP APIs not related to GUI development. Focus here is given to the persistent data, Media, Networking and Push Registry APIs.

2.4.1 Record Management System

MIDP provides a simple record-based persistent storage mechanism known as the Record Management System (RMS). It's a package that

allows the MIDlet application to store persistent data within a controlled environment, while maintaining system security. It provides a simple, non-volatile data store for MIDlets while they are not running. The classes making up the RMS are contained in the `javax.microedition.rms` package.

Essentially, the RMS is a very small, basic database. It stores binary data in a `Record` within a `RecordStore`. MIDlets can add, remove and update the records in a `RecordStore`. The persistent data storage location is implementation-dependent and is not exposed to the MIDlet. A `RecordStore` is, by default, accessible across all MIDlets within a suite, and MIDP extends access to MIDlets with the correct access permissions from other MIDlet suites. When the parent MIDlet suite is removed from the device, its record stores are also removed, regardless of whether a MIDlet in another suite is making use of them.

Here is a short example of how to use RMS for writing and reading persistent data from the device:

```
// saving data to RMS
public int saveToStore(byte[] data) {
  int recordID = 0;
  try {
    RecordStore store = RecordStore.openRecordStore("ImageStore", true);
    recordID = store.addRecord(data, 0, data.length);
    store.closeRecordStore();
  }
  catch(RecordStoreException rse) {
    rse.printStackTrace();
  }
  return recordID;
}
// reading data from RMS
public byte[] loadFromStore(String storeName, int recordID) {
  byte[] data = null;
  try {
    RecordStore store = RecordStore.openRecordStore("ImageStore", false);
    data = store.getRecord(recordID);
    store.closeRecordStore();
  }
  catch(RecordStoreException rse) {
    rse.printStackTrace();
  }
  return data;
}
```

For more about RMS, including searching and sorting records in a `RecordStore`, refer to the MIDP documentation. Also download the `RMSWriter` and `RMSReader` example applications from this book's website; they provide a fairly complete example of how to write, read and share `RecordStore`s.

2.4.2 Media API

MIDP includes the Media API which provides limited, audio-only multimedia. It is a subset of the optional and much richer JSR-135 Mobile Media API, which currently ships on most Symbian OS phones.

The MIDP Media API provides support for tone generation and audio playback of WAV files if the latter is supported by the underlying hardware. Since MIDP is targeted at the widest possible range of devices, not just feature-rich smartphones, the aim of the Media API is to provide a lowest common denominator of functionality suitable for the capabilities of all MIDP devices. However, with the wide availability of JSR-135 MMAPI on Symbian OS devices, we look at both the Media and Mobile Media APIs in detail later in this chapter, as the latter is much richer in functionality and provides additional opportunities for development of media-centric Java applications.

2.4.3 Networking – Generic Connection Framework

CLDC has defined a streamlined approach to networking, known as the Generic Connection Framework (GCF). The framework seeks to provide a consistent interface for every network connection between the MIDP classes and the underlying network protocols. It doesn't matter what kind of connection is being opened; the interface remains the same. For instance, if you're opening a socket or an HTTP connection, you are still going to use the same `Connector.open()` method. MIDP has support for many protocols, although HTTP and HTTPS are mandatory. Java ME on Symbian OS also supports other optional protocols, such as sockets, server sockets and datagrams. Due to the implications of the MIDP security model on networking APIs, these are discussed in more detail in Section 2.6.

2.4.4 Push Registry

The Push Registry API allows MIDlets to be launched in response to incoming network connections. Many applications, particularly messaging applications, need to be continuously listening for incoming messages. To achieve this in the past, a Java application would have had to be continually running in the background. Although the listening Java application may itself be small, it would still require an instance of the virtual machine to be running, thus appropriating some of the mobile phone's scarce resources. The JSR-118 group recognized the need for an alternative, more resource-effective solution for MIDP and so introduced the Push Registry. The Push Registry API is also affected by MIDP 2.0's security model, therefore we discuss it in Section 2.7.

2.5 MIDP Security Model

The MIDP security model is built on two concepts:

- Trusted MIDlet suites are those whose origin and integrity can be trusted by the device on the basis of some objective criterion.
- Protected APIs are APIs to which access is restricted, with the level of access being determined by the permissions (Allowed or User) allocated to the API.

A protection domain defines a set of permissions which grant, or potentially grant, access to an associated set of protected APIs. An installed MIDlet suite is bound to a protection domain, thereby determining its access to protected APIs.

An MIDP device must support at least one protection domain, the untrusted domain, and may support several protection domains, although a given MIDlet suite can only be bound to one protection domain. The set of protection domains supported by an implementation defines the security policy. If installed, an unsigned MIDlet suite is always bound to the untrusted domain, in which access to protected APIs may be denied or require explicit user permission. Since a requirement of the MIDP specification is that a MIDlet suite written to the MIDP 1.0 specification runs unaltered in an MIDP environment, MIDP 1.0 MIDlets are automatically treated as untrusted.

2.5.1 The X.509 PKI

The mechanism for identifying and verifying that a signed MIDlet suite should be bound to a trusted domain is not mandated by the MIDP specification but is left to the manufacturer of the device and other stakeholders with an interest in the security of the device, for example, network operators. The specification does, however, define how the X.509 Public Key Infrastructure (PKI) can be used to identify and verify a signed MIDlet suite.

The PKI is a system for managing the creation and distribution of digital certificates. At the heart of the PKI lies the system of public key cryptography. Public key cryptography involves the creation of a key pair consisting of a private key and a public key. The creator of the key pair keeps the private key secret, but can freely distribute the public key. Public and private key pairs have two principal uses: they enable secure communication using cryptography and authentication using digital signatures. In the first case, someone wishing to communicate with the holder of the private key uses the public key to encrypt the communication. The encrypted communication is secure since it can only be decrypted by the holder of the private key.

In the current context, however, we are more interested in the second use of public–private key pairs: enabling authentication using digital signatures. A digital signature is an electronic analogy of a conventional signature. It authenticates the source of the document and verifies that the document has not been tampered with in transit. Signing a document is a two-stage process: a message digest is created that is a unique representation of the contents of the document; the message digest is then encrypted using the private key of the sender (see Figure 2.11).

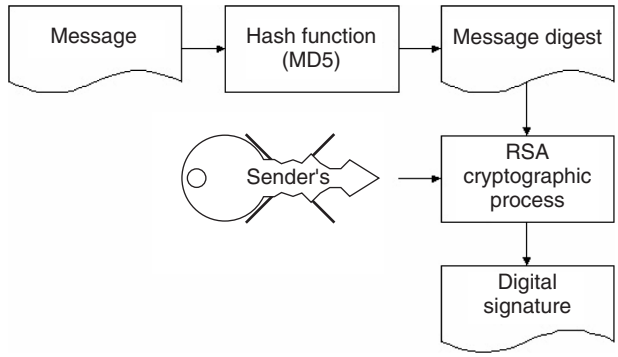

Figure 2.11 Encryption process in a nutshell

The receiver of the document then uses the public key of the sender to decrypt the message digest, creates a digest of the received contents, and checks that it matches the decrypted digest that accompanied the document. Hence, a digital signature is used to verify that a document was sent by the holder of the private key, not some third party masquerading as the sender, and that the contents have not been tampered with in transit. This raises the issue of key management and how the receiver of a public key can verify the source of the public key. For instance, if I receive a digitally signed JAR file, I need the public key of the signer to verify the signature, but how do I verify the source of the public key? The public key itself is just a series of numbers, with no clue as to the identity of the owner. I need to have confidence that a public key purporting to belong to a legitimate organization does in fact originate from that organization and has not been distributed by an impostor, enabling the impostor to masquerade as the legitimate organization, signing files using the private key of a bogus key pair. The solution is to distribute the public key in the form of a certificate from a trusted certificate authority (CA).

A certificate authority distributes a certificate that contains details of a person's or organization's identity, the public key belonging to that person or organization, and the identity of the issuing CA. The CA vouches that the public key contained in the certificate does indeed belong to the person or organization identified on the certificate. To verify that the

certificate was issued by the CA, the certificate is digitally signed by the CA using its private key. The format of certificates used in X509.PKI is known as the X509 format.

Of course, this raises the question of how the recipient of the certificate verifies the digital signature contained therein. This is resolved using root certificates or root keys. The root certificate contains details of the identity of the CA and the public key of the CA (the root key) and is signed by the CA itself (self-signed). For mobile phones that support one or more trusted protection domains, one or more certificates ship with the device, placed on the phone by the manufacturer or embedded in the WIM/SIM card by the network operator. Each certificate is associated with a trusted protection domain, so that a signed MIDlet that is authenticated against a certificate is bound to the protection domain associated with that certificate.

2.5.2 Certification Paths

In practice, the authentication of a signed file using the root certificate may be more complex than the simplified approach described above. The PKI allows for a hierarchy of certificate authorities (see Figure 2.12)

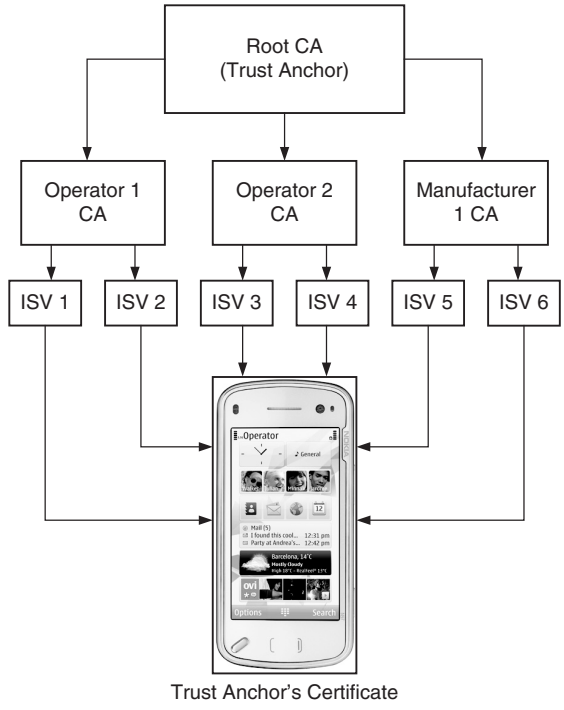

Figure 2.12 Applications from a variety of independent software vendors (ISVs) signed by various CAs and authenticated by a single trust root

whose validity can be traced back to a root certification authority, the uppermost CA in the hierarchy, also known as the trust anchor. In this case the root certificate on the device (the trust root) belongs to the root certification authority in the hierarchy (the trust anchor) which directly or indirectly validates all the other CAs in the certification path. The certificate supplied with the signed JAR file does not need to be validated (signed) by the trust anchor whose certificate is supplied with the device, as long as a valid certification path can be established between the certificate accompanying the signed JAR file and the root CA. It is not actually necessary for a device to have various self-signed top-level certificates from CAs, manufacturers and operators installed. In practice, it only needs access to one or more certificates which are known to be trustworthy, for example, because they are in ROM or secure storage on a WIM/SIM, or because the user has decided that they are.

These certificates act as trust roots. If the authentication of an arbitrary certificate chains back to a trust root known to the device, and the trust root is also identified as being suitable for authenticating certificates being used for a given purpose, for example, code-signing, website identification, and so on, then the arbitrary certificate is considered to have been authenticated.

2.5.3 Signing a MIDlet Suite

To sign a MIDlet suite, a supplier must create a public–private key pair and sign the MIDlet JAR file with the private key. The JAR file is signed using the RSA-SHA1 algorithm. The resulting signature is encoded in Base64 format and inserted into the application descriptor as the following attribute:

```
MIDlet-Jar-RSA-SHA1: <base64 encoding of JAR signature>
```

The supplier must obtain a suitable MIDlet suite code-signing certificate from an appropriate source, for example, the developer program of a device manufacturer or network operator, containing the identity of the supplier and the supplier's public key. The certificate is incorporated into the MIDlet suite's application descriptor (JAD) file.

In the case of a certification path, we need to include all the necessary certificates required to validate the JAR file. Furthermore, a MIDlet suite may include several certification paths in the application descriptor file (if, for example, the MIDlet suite supplier wishes to target several device types, each with a different root certificate). In Figure 2.13, we need to include certificates containing the public keys belonging to CA 1,

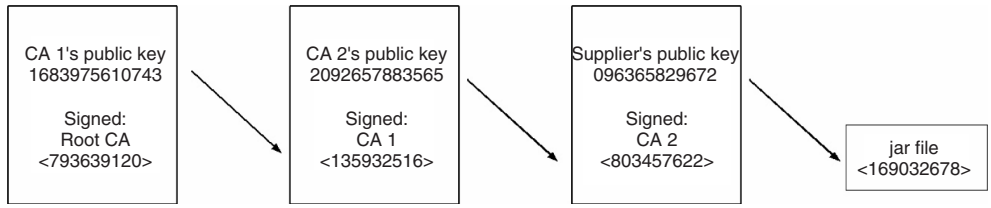

Figure 2.13 Certification path

CA 2 and the Supplier. The root certification authority's certificate (the root certificate) is available on the device. Using the root certification authority's public key, we can validate CA 1's public key. This is then used to validate CA 2's public key, which is then used to validate the Supplier's public key. The Supplier's public key is then used to verify the origin and integrity of the JAR file. The MIDP specification defines an application descriptor attribute of the following format:

```
MIDlet-Certificate-<n>-<m>: <base64 encoding of a certificate>
```

Here <n> represents the certification path and has a value of 1 for the first certification path, with each additional certification path adding 1 to the previous value (i.e. 1, 2, 3, ...). There may be several certification paths, each leading to a different root CA. <m> has a value of 1 for the certificate belonging to the signer of the JAR file and a value 1 greater than the previous value for each intermediate certificate in the certification path.

For the example shown in Figure 2.13, with just one certification path, the relevant descriptor attribute entries would have the following content:

```
MIDlet-Certificate-1-1: <base64 encoding of Supplier's certificate>
MIDlet-Certificate-1-2: <base64 encoding of CA 2's certificate>
MIDlet-Certificate-1-3: <base64 encoding of CA 1's certificate>
```

2.5.4 Authenticating a Signed MIDlet Suite

Before a MIDlet suite is installed, the Application Management Software (AMS) checks for the presence of the `MIDlet-Jar-RSA-SHA1` attribute in the application descriptor and, if it is present, attempts to authenticate the JAR file by verifying the signer certificate. If it is not possible to successfully authenticate a signed MIDlet suite, it is not installed. If the MIDlet suite descriptor file does not include the `MIDlet-Jar-RSA-SHA1` attribute, then the MIDlet can only be installed as untrusted.

2.5.5 Authorization Model

A signed MIDlet suite containing MIDlets which access protected APIs must explicitly request the required permissions. The MIDP specification defines two new attributes, `MIDlet-Permissions` and `MIDlet-Permissions-Opt`, for this purpose. Critical permissions (those that are required for MIDP access to protected APIs that are essential to the operation of MIDlets) must be listed under the `MIDlet-Permissions` attribute. Non-critical permissions (those required to access protected APIs without which the MIDlets can run in a restricted mode) should be listed under the `MIDlet-Permissions-Opt` attribute.

The `MIDlet-Permissions` and `MIDlet-Permissions-Opt` attributes may appear in the JAD file or the manifest of a signed MIDlet suite, or in both, in which case their respective values in each must be identical, but only the values in the manifest are 'protected' by the signature of the JAR file.

It is important to note that a MIDlet suite that has been installed as trusted is not granted any permission it has not explicitly requested in either the `MIDlet-Permissions` or `MIDlet-Permissions-Opt` attributes, irrespective of whether it would be granted were it to be requested.

The naming scheme for permissions is similar to that for Java package names. The exact name of a permission to access an API or function is defined in the specification for that API. For instance, an entry requesting permission to open HTTP and secure HTTP connections would be as follows:

```
MIDlet-Permissions: javax.microedition.io.Connector.http,
                    javax.microedition.io.Connector.https
```

The successful authorization of a trusted MIDlet suite requires that the requested critical permissions are recognized by the device (for instance, in the case of optional APIs) and are granted, or potentially granted, in the protection domain to which the MIDlet suite would be bound, were it to be installed. If either of these requirements cannot be satisfied, the MIDlet suite is not installed.

2.5.6 Protection Domains

A protection domain is a set of permissions determining access to protected APIs or functions. A permission is either Allowed, in which case MIDlets in MIDlet suites bound to this protection domain have automatic access to this API, or User, in which case permission to access the protected API or function is requested from the user, who can then grant or

deny access. In the case of User permissions, there are three interaction modes:

- `Blanket` – as long as the MIDlet suite is installed, it has this permission unless the user explicitly revokes it.
- `Session` – user authorization is requested the first time the API is invoked and it is in force while the MIDlet is running.
- `Oneshot` – user authorization is requested each time the API is invoked.

The protection domains for a given device are defined in a security policy file. A sample security policy file is shown below:

```
alias: net_access
javax.microedition.io.Connector.http,
javax.microedition.io.Connector.https,
javax.microedition.io.Connector.datagram,
javax.microedition.io.Connector.datagramreceiver,
javax.microedition.io.Connector.socket,
javax.microedition.io.Connector.serversocket,
javax.microedition.io.Connector.ssl
domain: Untrusted
session (oneshot): net_access
oneshot (oneshot): javax.microedition.io.Connector.sms.send
oneshot (oneshot): javax.microedition.io.Connector.sms.receive
session (oneshot): javax.microedition.io.PushRegistry
domain: Symbian
allow: net_access
allow: javax.microedition.io.Connector.sms.send
allow: javax.microedition.io.Connector.sms.receive
allow: javax.microedition.io.PushRegistry
```

User permissions may offer several interaction modes, the user being able to select the level of access. For instance, the following line indicates that the API or functions defined under the `net_access` alias have User permission with either `session` or `oneshot` interaction modes, the latter being the default:

```
session (oneshot): net_access
```

2.5.7 The Security Model in Practice

In this section, we go through the steps involved in producing a signed MIDlet suite. We shall illustrate this process using the tools provided by the WTK. The basic steps in producing a signed MIDlet suite are listed below:

1. Obtain (or generate) a public–private key pair.

2. Associate the key pair with a code-signing certificate from a recommended CA.
3. Sign the MIDlet suite and incorporate the certificate into the JAD file.

To sign a MIDlet suite, the supplier of the suite needs to obtain a public–private key pair either by generating a new key pair or importing an existing key pair. The WTK provides tools for doing this; they can be accessed by opening your project and choosing the Project/Sign option from the main panel. Clicking the Sign button brings up the panel shown in Figure 2.14. To generate the key pair, click on Keystore, then New Key Pair, enter the appropriate details and click the Create button (see Figure 2.15).

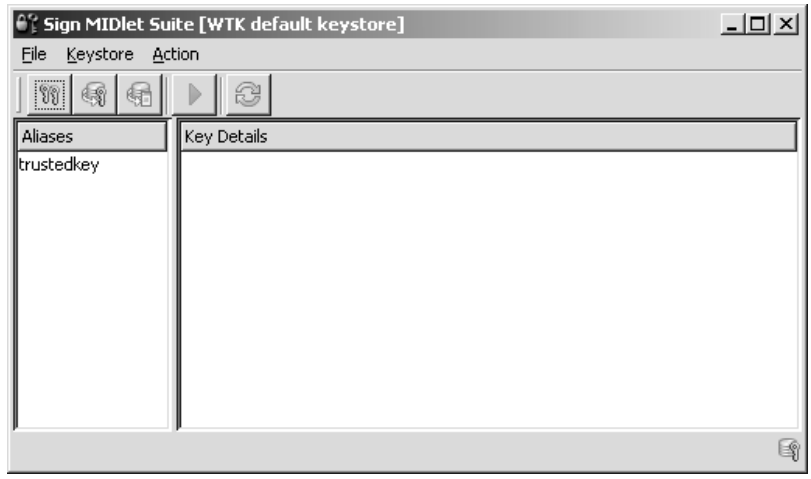

Figure 2.14 Sign MIDlet Suite view of the WTK

A new key pair is generated and added to the WTK key store. The newly-generated public key is incorporated into a self-signed certificate. We use this to obtain a suitable MIDlet suite code-signing certificate from an appropriate source (such as a recommended Certification Authority, for instance, Verisign or Thawte) that can be authenticated by a root certificate that ships with the device or is contained in the WIM/SIM card. Application developers and suppliers should contact the relevant developer program of the device manufacturer or network operator to ascertain the appropriate CA.

We can then generate a Certificate Signing Request (CSR) using our self-signed certificate and the Generate CSR option in the WTK (Figure 2.16). This generates a file containing the CSR that can be saved to a convenient location. The contents of the CSR can then be copied into an email to the recommended CA, requesting a code-signing certificate.

Figure 2.15 Creating a new key pair

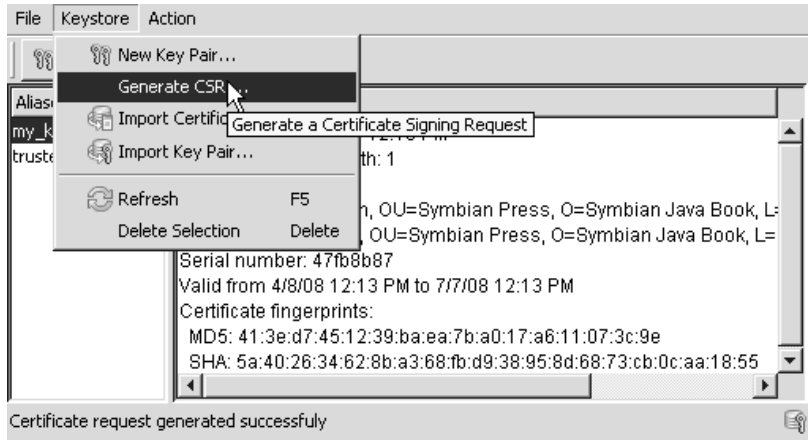

Figure 2.16 Generating a Certificate Signing Request

When we have received the certificate from the recommended CA, we need to associate this with our key pair. The Import Certificate option of the WTK associates the certificate with our key pair, identified by its alias and held in the key store. If the public key that we provided in the CSR, and now contained in the certificate, matches the public key of the key pair held in the key store, we should be notified accordingly and are now ready to sign our MIDlet suite. To sign the MIDlet suite we simply select Sign MIDlet Suite from the Action menu and choose the JAD file

belonging to the MIDlet suite we wish to sign. The MIDlet suite is now ready for deployment on the target device.

The WTK also offers additional functionality to test out signed MIDlet suites. There is a default trusted key pair (and an associated root certificate) that can be used to install and bind a signed MIDlet suite to a trusted protection domain within the emulator environment. This then allows MIDlets in the signed suite to run within the WTK environment as trusted without obtaining and importing a certificate from a CA. This feature is particularly useful for ensuring that the appropriate permissions to access protected APIs have been requested in the JAD file. It is important to remember that this feature of the WTK only works in the test environment. For real devices, you must sign your MIDlet suite with a valid certificate received from a Trusted CA.

2.5.8 Untrusted MIDlets

An untrusted MIDlet suite is an unsigned MIDlet suite. It is, therefore, installed and bound to the untrusted protection domain. Untrusted MIDlets execute within a restricted environment where access to protected APIs or functions may be prohibited or allowed only with explicit user permission, depending on the security policy in force on the device. To ensure compatibility with MIDlets developed according to the MIDP 1.0 specification, the MIDP specification demands that the untrusted domain must allow unrestricted access to the following APIs:

- `javax.microedition.rms`
- `javax.microedition.midlet`
- `javax.microedition.lcdui`
- `javax.microedition.lcdui.game`
- `javax.microedition.media`
- `javax.microedition.media.control`

Furthermore, the specification requires that the following APIs can be accessed with explicit permission of the user:

- `javax.microedition.io.HttpConnection`
- `javax.microedition.io.HttpsConnection`

The full list of permissions for the untrusted domain is device-specific, however the MIDP specification does provide a Recommended Security Policy Document for GSM/UMTS Compliant Devices as an addendum (with some clarifications added in the JTWI Final Release Policy for

Untrusted MIDlet Suites). Finally, if a signed MIDlet fails authentication or authorization, it does not run as an untrusted MIDlet, but rather is not installed by the AMS. For more information on the security model, see the MIDP specification.

2.5.9 Recommended Security Policy

The Recommended Security Policy defines a set of three protection domains (Manufacturer, Operator and Trusted Third Party) to which trusted MIDlet suites can be bound (and which a compliant device may support).

For a trusted domain to be enabled, there must be a certificate on the device, or on a WIM/SIM, identified as a trust root for MIDlet suites in that domain, i.e. if a signed MIDlet suite can be authenticated using that trust root it will be bound to that domain. For example, to enable the Manufacturer protection domain, the manufacturer must place a certificate on the device. This is identified as the trust root for the Manufacturer domain. A signed MIDlet suite will be bound to the Operator domain if it can be authenticated using a certificate found on the WIM/SIM and identified as a trust root for the Operator domain. A signed MIDlet suite will be bound to the Trusted Third Party protection domain if it can be authenticated using a certificate found on the device or on a WIM/SIM and identified as a trust root for the Trusted Third Party protection domain. Verisign and Thawte code-signing certificates usually fit the latter situation.

As already mentioned, the recommended security policy is not a mandatory requirement for an MIDP 2.0-compliant device. An implementation does not have to support the RSP in order to install signed MIDlet suites; it simply has to implement the MIDP security model and support at least one trusted protection domain.

2.6 Networking and General Connection Framework

Now that we have considered in detail the MIDP Security Model, let's learn more about the networking APIs in the specification, provided through the Generic Connection Framework (GCF).

Symbian's implementation of MIDP complies with the specification, providing implementations of the following protocols:

- HTTP
- HTTPS
- Sockets

- Server sockets
- Secure sockets
- Datagrams
- Serial port access.

In the remainder of this chapter, we focus on the networking protocols from the GCF. The local connectivity protocol (serial port access) is similar in nature to the network protocols, but its connection string depends on the target handset model, therefore you must study it separately.

2.6.1 HTTP and HTTPS Support

To open an HTTP connection we use the `Connector.open()` method with a URL of the form ***www.myserver.com***. Code to open an `HttpConnection` and obtain an `InputStream` would look something like this:

```
try{
  String url = "www.myserver.com";
  HttpConnection conn = (HttpConnection)Connector.open(url);
  InputStream is = conn.openInputStream();
  ...
  conn.close()
}
catch(IOException ioe){...}
```

Under the MIDP security model, untrusted MIDlets can open an HTTP connection only with explicit user confirmation. Signed MIDlets that require access to an HTTP connection must explicitly request permission:

```
MIDlet-Permissions: javax.microedition.io.Connector.http, ...
```

Opening an HTTPS connection follows the same pattern as a normal HTTP connection, with the exception that we pass in a connection URL of the form ***https://www.mysecureserver.com*** and cast the returned instance to an `HttpsConnection` object, as in the following example:

```
try{
  String url = "https://www.mysecureserver.com";
  HttpsConnection hc = (HttpsConnection)Connector.open(url);
  InputStream is = hc.openInputStream();
  ...
  hc.close()
```

```
  ...
}
catch(IOException ioe){...}
```

The security policy related to access permissions by trusted and untrusted MIDlets is the same as the one for HTTP connections, except that the permission setting has a different name:

```
MIDlet-Permissions: javax.microedition.io.Connector.https, ...
```

2.6.2 Socket and Server Socket Support

MIDP makes support for socket connections a recommended practice. Socket connections come in two forms: client connections, in which a socket connection is opened to another host; and server connections in which the system listens on a particular port for incoming connections from other hosts. The connections are specified using Universal Resource Identifiers (URI). You should be familiar with the syntax of a URI from web browsing. They have the format `<string1>://<string2>` where `<string1>` identifies the communication protocol to be used (e.g., `http`) and `<string2>` provides specific details about the connection. The protocol may be one of those supported by the Generic Connection Framework (see Section 2.5.2).

To open a client socket connection to another host, we pass a URI of the following form to the connector's `open()` method:

```
socket://www.symbian.com:5000
```

The host may be specified as a fully qualified hostname or IP address and the port number refers to the connection endpoint on the remote peer. Some sample code is shown below:

```
SocketConnection sc = null;
OutputStream out = null;
try{
  sc = (SocketConnection)Connector.open ("socket://200.251.191.10:7900");
  ...
  out = c.openOutputStream();
  ...
}
catch(IOException ioe){...}
```

A server socket connection is used to listen for inbound socket connections. To obtain a server socket connection, we can pass a URI in either of the following forms to the connector's `open()` method:

```
socket://:79
socket://
```

In the first case, the system listens for incoming connections on port 79 (of the local host). In the latter case, the system allocates an available port for the incoming connections.

```
ServerSocketConnection ssc = null;
InputStream is = null;
try{
  ssc = (ServerSocketConnection)Connector.open("socket://:1234");
  SocketConnection sc = (SocketConnection)ssc.acceptAndOpen();
  ...
  is = sc.openInputStream();
  ...
}
catch(IOException ioe){...}
```

The `ServerSocketConnectioninterface` extends the `StreamConnectionNotifier` interface. To obtain a connection object for an incoming connection the `acceptAndOpen()` method must be called on the `ServerSocketConnection` instance. An inbound socket connection results in the call to the `acceptAndOpen()` method, returning a `StreamConnection` object which can be cast to a `SocketConnection` as desired.

A signed MIDlet suite containing MIDlets which open socket connections must explicitly request the appropriate permissions:

```
MIDlet-Permissions: javax.microedition.io.Connector.socket,
              javax.microedition.io.Connector.serversocket, ...
```

If the protection domain to which the signed MIDlet suite would be bound grants, or potentially grants, these permissions, then the MIDlet suite is installed and the MIDlets it contains will be able to open socket connections. This can be done either automatically or with user permission, depending upon the security policy in effect on the device for the protection domain to which the MIDlet suite has been bound.

Whether MIDlets in untrusted MIDlet suites can open socket connections depends on the security policy relating to the untrusted domain in force on the device.

2.6.3 Secure Socket Support

Secure socket connections are client socket connections over SSL. To open a secure socket connection we pass in a hostname (or IP address)

and port number to the connector's `open()` method using the following URI syntax:

```
ssl://hostname:port
```

We can then use the secure socket connection in the same manner as a normal socket connection, for example:

```
try{
  SecureConnection sc = (SecureConnection)
                        Connector.open("ssl://www.secureserver.com:443");
  ...
  OutputStream out = sc.openOutputStream();
  ...
  InputStream in = sc.openInputStream();
  ...
}
catch(IOException ioe){...}
```

A signed MIDlet suite that contains MIDlets which open secure connections must explicitly request permission:

```
MIDlet-Permissions: javax.microedition.io.Connector.ssl, ...
```

If the protection domain to which the signed MIDlet suite would be bound grants, or potentially grants, this permission, the MIDlet suite can be installed and the MIDlets it contains will be able to open secure connections. This can be done automatically or with user permission, depending on the security policy in effect. Whether untrusted MIDlets can open secure connections depends on the permissions granted in the untrusted protection domain.

2.6.4 Datagram Support

Symbian's MIDP implementation includes support for sending and receiving UDP datagrams. A datagram connection can be opened in client or server mode. Client mode is for sending datagrams to a remote device. To open a client-mode datagram connection we use the following URI format:

```
datagram://localhost:1234
```

Here the port number indicates the port on the target device to which the datagram will be sent. Sample code for sending a datagram is shown below:

```
String message = "Hello!";
```

```
byte[] payload = message.toString();
try{
  UDPDatagramConnection conn = null;
  conn = (UDPDatagramConnection)
                              Connector.open("datagram://localhost:1234");
  Datagram datagram = conn.newDatagram(payload, payload.length);
  conn.send(datagram);
}
catch(IOException ioe){...}
```

Server mode connections are for receiving (and replying to) incoming datagrams. To open a datagram connection in server mode, we use a URI of the following form:

```
datagram://:1234
```

The port number in this case refers to the port on which the local device is listening for incoming datagrams. Sample code for receiving incoming datagrams is given below:

```
  try{
  UDPDatagramConnection dconn = null;
  dconn = (UDPDatagramConnection)Connector.open("datagram://:1234");
  Datagram dg = dconn.newDatagram(300);
  while(true){
    dconn.receive(dg);
    byte[] data = dg.getData();
    ...
  }
}
catch(IOException ioe){...}
```

A signed MIDlet suite which contains MIDlets that open datagram connections must explicitly request permission to open client connections or server connections:

```
MIDlet-Permissions: javax.microedition.io.Connector.datagram,
            javax.microedition.io.Connector.datagramreceiver, ...
```

If the protection domain to which the signed MIDlet suite would be bound grants, or potentially grants, the requested permissions, the MIDlet suite can be installed and the MIDlets it contains will be able to open datagram connections. This can be done automatically or with user permission, depending on the security policy in effect. Whether untrusted MIDlets can open datagram connections depends on permissions granted to MIDlet suites bound to the untrusted protection domain.

2.6.5 Security Policy for Network Connections

The connections discussed above are part of the Net Access function group (see the RSP addendum to the MIDP specification). On the Nokia N95, for example, an untrusted MIDlet can access the Net Access function group with User permission (explicit confirmation required from the user). Figure 2.17 shows an example of an unsigned MIDlet, Google's Gmail, and the available permission options on a Nokia N95. This policy varies from licensee to licensee so you must check with the manufacturer of your target devices which settings apply for existing security domains.

Figure 2.17 Permission options for an unsigned MIDlet on a Nokia N95

2.6.6 `NetworkDemo` MIDlet

We finish this section with a simple example using MIDP's `javax.microedition.io.HttpConnection` to interrogate a web server. The `NetworkDemo` MIDlet connects to a web server via `HttpConnection`, and reads and displays all the headers and the first 256 characters of the response itself. Let's take a look at the core network functionality exploited in this MIDlet. The connection work is all done in the `ClientConnection` class:

```
public class ClientConnection extends Thread {
  private static final int TEXT_FIELD_SIZE = 256;
  private NetworkDemoMIDlet midlet = null;
  private String url = null;
```

```
  public ClientConnection(NetworkDemoMIDlet midlet) {
    this.midlet = midlet;
  }
  public void sendMessage(String url) {
    this.url = url;
    start();
  }
  public void run() {
    try{
      HttpConnection conn = (HttpConnection)Connector.open(url);
      int responseCode = conn.getResponseCode();

      midlet.append("Response code","" + responseCode);
      int i = 0;
      String headerKey = null;
      while((headerKey = conn.getHeaderFieldKey(i++)) != null)  {
        midlet.append(headerKey,conn.getHeaderField(headerKey));
      }

      InputStream in = conn.openInputStream();
      StringBuffer buffer = new StringBuffer(TEXT_FIELD_SIZE);
      int ch;
      int read = 0;

      while ( (ch = in.read()) != -1 && read < TEXT_FIELD_SIZE) {
        buffer.append((char)ch);
        read++;
      }

      midlet.append("HTML Response" ,buffer.toString());
      conn.close();
      conn = null;
    }
    catch(Exception e){
      e.printStackTrace();
    }
  }
}
```

The `url` parameter of the `sendMessage()` method has the following form:

```
http://www.symbian.com:80
```

The `sendMessage()` method creates a request and then starts a new `Thread` to create the connection, send the request and read the response. Let us look at the contents of the thread's `run()` method in more detail:

```
HttpConnection conn = (HttpConnection)Connector.open(url);
int responseCode = conn.getResponseCode();
midlet.append("Response code","" + responseCode);
int i = 0;
```

```
String headerKey = null;
while((headerKey = conn.getHeaderFieldKey(i++)) != null)  {
  midlet.append(headerKey,conn.getHeaderField(headerKey));
}
```

An `HttpConnection` is opened using the URI given by the user and is used first to retrieve the HTTP response code (200, in case of success). We also loop through the HTTP response methods until they are all read and append them to the MIDlet's `Form` object as a `StringItem`.

Having read the HTTP readers, it is time to read the actual HTML content. We open an `InputStream` from the `HttpConnection`, start reading the input and save it to a `StringBuffer` of TEXT_FIELD_SIZE length. Once we reach this value in number of bytes read, we leave the loop, as we don't want to pollute the user's screen with too much raw HTML.

```
InputStream in = conn.openInputStream();
StringBuffer buffer = new StringBuffer(TEXT_FIELD_SIZE);
int ch;
int read = 0;
while ( (ch = in.read()) != -1 && read < TEXT_FIELD_SIZE) {
  buffer.append((char)ch);
  read++;
}
```

After reading TEXT_FIELD_SIZE bytes of HTML, we append them to the MIDlet's `Form` so they can be shown on the screen along with all the other information we retrieved. Figure 2.18 shows the `NetworkDemo` MIDlet running on a Nokia N95. The full source code, JAD and JAR files for the `NetworkDemo` MIDlet are available for download from this book's website.

(a) (b) (c)

Figure 2.18 `NetworkDemo` MIDlet running on a Nokia N95: a) connecting to the URL, b) receiving the headers, and c) reading the HTML

2.7 Using the Push Registry

The Push Registry API is encapsulated in the `javax.microedition.io.PushRegistry` class. It maintains a list of inbound connections that have been previously registered by installed MIDlets. A MIDlet registers an incoming connection with the push registry either statically at installation via an entry in the JAD file or dynamically (programmatically) via the `registerConnection()` method.

When a MIDlet is running, it handles all the incoming connections (whether registered with the push registry or not). When the MIDlet is not running, the AMS launches the MIDlet in response to an incoming connection previously registered by it, by invoking the `startApp()` method. The AMS then hands off the connection to the MIDlet which is responsible for opening the appropriate connection and handling the communication. In the case of static registration, the MIDlet registers its interest in incoming connections in the JAD file, in the following format:

```
MIDlet-Push-<n>: <ConnectionURL>, <MIDletClassName>, <AllowedSender>
```

The `<ConnectionURL>` field specifies the protocol and port for the connection end point in the same URI syntax used by the argument to the `Connector.open()` method that is used by the MIDlet to process the incoming connection. Examples of `<ConnectionURL>` entries might be:

```
sms://:1234
socket://:1234
```

The `<MIDletClassName>` field contains the package-qualified name of the class that extends `javax.microedition.midlet.MIDlet`. This would be the name of the MIDlet class as listed in the application descriptor or manifest file under the `MIDlet-<n>` entry. The `<AllowedSender>` field acts as a filter indicating that the AMS should only respond to incoming connections from a specific sender. For the SMS protocol, the `<AllowedSender>` entry is the phone number of the required sender. For a server socket connection end-point, the `<AllowedSender>` entry would be an IP address (in both cases, note that the sender port number is not included in the filter). The `<AllowedSender>` syntax supports two wildcard characters: * matches any string including an empty string and ? matches any character. Hence the following would be valid entries for the `<AllowedSender>` field:

```
*
129.70.40.*
129.70.40.23?
```

The first entry indicates any IP address, the second entry allows the last three digits of the IP address to take any value, while the last entry allows only the last digit to have any value. So the full entry for the `MIDlet-Push-<n>` attribute in a JAD file may look something like this:

```
MIDlet-Push-1: sms://:1234, com.symbian.devnet.ChatMIDlet, *
MIDlet-Push-2: socket://:3000, com.symbian.devnet.ChatMIDlet, 129.70.40.*
```

If the request for a static connection registration cannot be fulfilled then the AMS must not install the MIDlet. Examples of when a registration request might fail include the requested protocol not being supported by the device, or the requested port number being already allocated to another application.

To register a dynamic connection with the AMS we use the static `registerConnection()` method of `PushRegistry`:

```
PushRegistry.registerConnection("sms://:1234",
                                    "com.symbian.devnet.ChatMIDlet", "*");
```

The arguments take precisely the same format as those used to make up the `MIDlet-Push-<n>` entry in a JAD or manifest. Upon registration, the dynamic connection behaves in an identical manner to a static connection registered via the application descriptor. To un-register a dynamic connection, the static Boolean `unregisterConnection()` method of `PushRegistry` is used:

```
boolean result = PushRegistry.unregisterConnection(("sms://:1234");
```

If the dynamic connection is successfully unregistered, a value of true is returned. The AMS responds to input activity on a registered connection by launching the corresponding MIDlet (assuming that the MIDlet is not already running). The MIDlet responds to the incoming connection by launching a thread to handle the incoming data in the `startApp()` method. Using a separate thread is recommended practice for avoiding conflicts between blocking I/O operations and normal user interaction events. For a MIDlet registered for incoming SMS messages, the `startApp()` method might look something like this:

```
public void startApp() {
  // List of active connections.
  String[] connections = PushRegistry.listConnections(true);
  for (int i=0; i < connections.length; i++) {
    if(connections[i].equals("sms://:1234")){
      new Thread(){
        public void run(){
          Receiver.openReceiver();
        }
      }.start();
    }
  }
  ...
}
```

One other use of the push registry should be mentioned before we leave this topic. The `PushRegistry` class provides a method which allows a running MIDlet to register itself or another MIDlet in the same suite for activation at a given time:

```
public static long registerAlarm(String midlet, long time)
```

The `midlet` argument is the class name of the MIDlet to be launched at the time specified by the `time` argument. The launch time is specified in milliseconds since January 1, 1970, 00:00:00 GMT. The push registry may contain only one outstanding activation time entry per MIDlet in each installed MIDlet suite. If a previous activation entry is registered, it is replaced by the current invocation and the previous value is returned. If no previous wakeup time has been set, zero is returned.

The `PushRegistry` is a protected API; a signed MIDlet suite which registers connections statically or contains MIDlets which register connections or alarms must explicitly request permission:

```
MIDlet-Permissions: javax.microedition.io.PushRegistry, ...
```

Note that a signed MIDlet suite must also explicitly request the permissions necessary to open the connection types of any connections it wishes to register. If the protection domain to which the signed MIDlet suite would be bound grants, or potentially grants, the requested permission, the MIDlet suite can be installed and the MIDlets it contains can register and deregister connections and alarms, either automatically or with user permission, depending on the security policy in effect.

Untrusted MIDlets do not require a `MIDlet-Permissions` entry. Whether access is granted to the Push Registry API depends on the security policy for the untrusted protection domain in effect on the device.

On the Nokia N95, untrusted MIDlet suites can use the Push Registry APIs (Application Auto-Start function group) with user permission. The default User permission is set to Session ('Ask first time'). It can be changed to 'Not allowed' or 'Ask every time'.

2.8 MIDP and the JTWI

The Java Technology for the Wireless Industry (JTWI) initiative is part of the Java Community Process (JSR-185) and its expert group has as its goal the task of defining an industry-standard Java platform for mobile phones, by reducing the need for proprietary APIs and providing a clear specification that phone manufacturers, network operators and developers can target. The JTWI specification concerns three main areas:

- It provides a minimum set of APIs (JSRs) that a compliant device should support.
- It defines which optional features within these component JSRs must be implemented on a JTWI-compliant device.
- It provides clarification of component JSR specifications, where appropriate.

2.8.1 Component JSRs of the JTWI

The JTWI defines three categories of JSR that fall under the specification: mandatory, conditionally required and minimum configuration. The following mandatory JSRs must be implemented as part of a Java platform that is compliant with JTWI:

- MIDP (JSR-118)
- Wireless Messaging API (JSR-120).

The Mobile Media API (JSR-135) is conditionally required in the JTWI. It must be present if the device exposes multimedia APIs (e.g., audio or video playback or recording) to Java applications. The minimum configuration required for JTWI compliance is CLDC 1.0 (JSR-30). Since CLDC 1.1 is a superset of CLDC 1.0 it may be used instead, in which case it supersedes the requirement for CLDC 1.0. Today most of the Symbian OS devices in the market support CLDC 1.1.

2.8.2 JTWI Specification Requirements

As mentioned earlier, the JTWI specification makes additional requirements on the implementation of the component JSRs. For full details,

consult the JTWI specification available from the Java Community Process (JCP) website (***http://jcp.org***).

- CLDC 1.0/1.1 must allow a MIDlet suite to create a minimum of 10 running threads and must support Unicode characters.
- MIDP 2.0 must allow creation of at least five independent record stores, must support the JPEG image format and must provide a mechanism for selecting a phone number from the device's phonebook when the user is editing a `TextField` or `TextBox` with the PHONENUMBER constraint.
- GSM/UMTS phones must support SMS protocol push handling within the Push Registry.
- MMA must support MIDI playback (with volume control), JPEG encoding for video snapshots and the Tone Sequence file format.

The JTWI specification also clarifies aspects of the MIDP recommended security policy for GSM/UMTS devices relating to untrusted domains.

2.8.3 Symbian and the JTWI

Symbian supports and endorses the efforts of the JTWI and is a member of the JSR-185 expert group. Releases of Symbian OS from version 9.1 provide implementations of the mandatory JSRs (Wireless Messaging API and Mobile Media API) and configurations (CLDC 1.0/1.1 and MIDP 2.0) specified by the JTWI. They also provide many other JSR implementations, about which you can find information on Symbian's website.

In 2003, JTWI was still in its draft phase, therefore only a couple of Symbian OS phones supported it, and only partially. At the time of writing (2008), the vast majority of Symbian devices supports the full range of JTWI specification plus many optional APIs. Developers can therefore rely on JSR-185 support on Symbian devices for their JTWI-based applications.

2.9 Mobile Media API

The Media API is fairly limited, giving support only to basic audio types. Smartphones these days have many more multimedia capabilities. Therefore it is necessary to expose all this multimedia functionality, such as tone generation, photo capture, and video playback and capture, to Java ME applications, so developers can create truly compelling multimedia applications attractive to end users.

This was achieved through the creation of JSR-135 – Mobile Media API (MMAPI). MMAPI is indicated for a JTWI-compliant device. Note that

it does not form part of the MIDP 2.0 specification; its specification in JSR-135 is presided over by an expert group.

Due to the wide scope encompassed by the word 'multimedia', MMAPI is very flexible and modular; it supports many different devices with different multimedia capabilities in a simple, understandable way for the developer. There are essentially three media types which can be handled within the framework: audio, video and generated tones. Any or all of these may be supported in a particular MMAPI implementation. Three tasks can be performed in relation to audio and video:

- playing – stored video or audio content is recovered from a file at a specified URI (or perhaps stored locally) and displayed onscreen or sent to a speaker
- capturing – a video or audio stream is obtained directly from hardware (camera or microphone) associated with the device
- recording – video or audio content which is being played or captured is sent to a specified URI or cached locally and made available for 're-playing'.

Implementations are not required to support all three tasks, although clearly it is not possible to support recording without supporting either playing or capturing. Further, in the context of a mobile phone, there is little point in supporting capture without the ability to play. So the practical options for audio and video are:

- only playing is supported
- playing and capturing are supported, but not recording
- playing and recording are supported, but not capturing
- all three are supported.

As we shall see, Symbian OS phones typically support all three functions. There are further choices in terms of supported formats and the ability to manipulate data streams or playback.

2.9.1 MMAPI Architecture

At the heart of MMAPI is the concept of a media player. Players are obtained from a factory or manager which also serves to associate them with a particular media stream. While the player allows basic start and stop capability for the playback, fine-grained manipulation, such as capturing and recording, is achieved through various kinds of controls, which are typically obtained from a player. Also of interest is the concept

of a player listener, which allows you to track the progress of players being initialized, started or stopped.

The following classes or interfaces embody the above basic concepts and are the fundamental elements of the core package `javax.microedition.media`:

- `Player`
- `Manager`
- `Control`
- `PlayerListener`.

In fact, of these only `Manager` is a concrete class; the others are all interfaces. The `Control` interface exists merely as a marker for the set of (more concrete) sub-interfaces which are defined in the related package `javax.microedition.media.control`. The functionality provided by the latter is so diverse that nothing is in common, and the `Control` interface, remarkably, defines no methods!

MMAPI also allows you the flexibility to define your own protocols for downloading or obtaining the media content to be played. This involves defining concrete implementations of the abstract `DataSource` class. As this is a specialist topic beyond the scope of this chapter, we shall not say anything further. The MMAPI specification document contains further details.

2.9.2 Obtaining Media Content

The elements of the Mobile Media API work together as shown in Figure 2.19 (the `PlayerListener` has been omitted and we shall return to it in a later section).

A `Player` is typically created using the following factory method of the `javax.microedition.media.Manager` class:

```
public static Player createPlayer(String locator)
```

The method's argument is a media locator, a string representing a URI that provides details about the media content being obtained. The details are specified using the well-documented Augmented Backus–Naur Format (ABNF) syntax. Table 2.2 lists some examples. MMAPI supports both communication and local protocols, which allow retrieval of the desired media from its source.

There are other variants of the `createPlayer()` method that take different parameters, such as an `InputStream` and a MIME type (for media stored locally) or a custom `DataSource` implemented by the

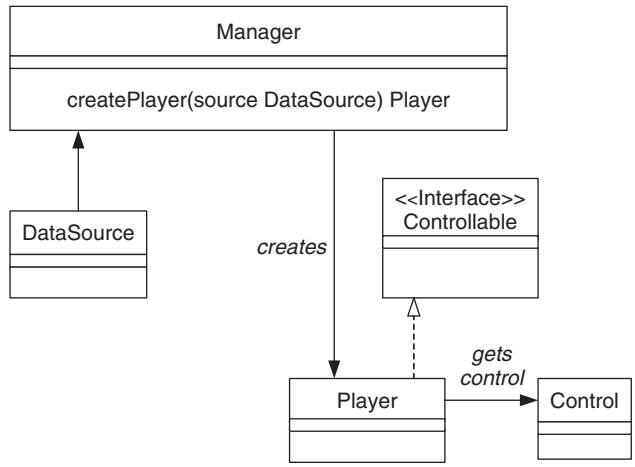

Figure 2.19 The basic architecture of the Mobile Media API

Table 2.2 Examples of Media Locator

Locator	Purpose	Examples
`capture://`	Retrieving live media data from hardware	`capture://audio` – captures microphone audio `capture://video` – captures camera output `capture://audio_video` – captures audio and video simultaneously (for some devices, only `capture://video` is necessary)
`device://`	Configuring players for tone sequences or MIDI data	`device://tone` – tone device `device://midi` – midi device
`rtsp://` `rtp://`	Media streaming, where playback starts before download is complete	`rtsp://streamer.whyy.org/encoder/live.rm`
`http://`	Fetching media stored on a web server	`http://www.yourhost.com/myfile.mp3`

developer. Please refer to the SDK documentation for more details on obtaining media using the player creation methods of the `Manager` class.

2.9.3 Playing Media Content

Because of the complexity of what a `Player` does, there are necessarily several stages (or states) it has to go through before it can play (see Figure 2.20). As we have just seen, the first stage is the creation of a `Player` object via the `Manager` factory class. This involves the creation of a `DataSource` object, which provides a standard interface to the media content. Unless you are working with a custom `DataSource`, the creation is done on your behalf. It's worth noticing that creating the `Player` object does not initialize the data transfer, which begins at the next stage.

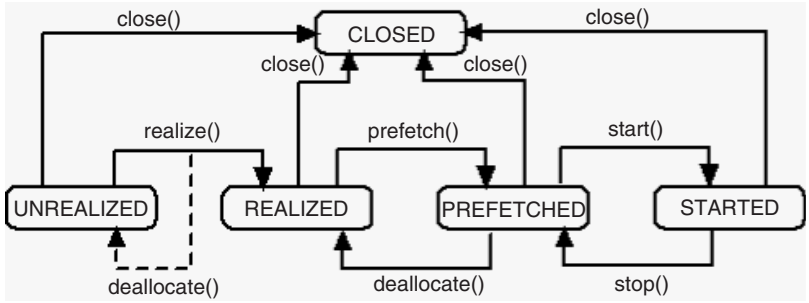

Figure 2.20 Lifecycle of a `Player` object

On creation, the `Player` is in the UNREALIZED state. Calling the `realize()` method causes the `Player` to initiate data transfer, for example, communicating with a server or a file system. Peer classes to marshal the data on the native side are typically instantiated at this point. When this method returns, the player is in the REALIZED state. Calling `prefetch()` causes the `Player` to acquire the scarce and exclusive resources it needs to play the media, such as access to the phone's audio device. It may have to wait for another application to release these resources before it can move to the PREFETCHED state. Once in this state, the `Player` is ready to start. A call to its `start()` method initiates playing and moves it on to the STARTED state. In order to interrogate the state of the `Player`, the `getState()` method of the `Player` class is provided.

In many cases, of course, clients of MMAPI will not be interested in the fine distinctions of which resources are acquired by which methods. The good news is that you are free to ignore them if you wish: a call to `start()` on a `Player` in any state other than CLOSED results in any

intermediate calls needed to `realize()` or `prefetch()` being made implicitly. Of course, the price you pay is less fine-grained control of exception handling.

The matching methods to stop the `Player` are `close()`, `deallocate()` and `stop()`. As with the `start()` method, the `close()` method encompasses the other two, so they need not be invoked on a `Player` directly. You should be aware, however, that reaching the end of the media results in the `Player` returning to the PREFETCHED state, as though the `stop()` method had been called. The good thing about this is that you can then conveniently replay the media by calling `start()` again. However, you must call the `close()` method explicitly to recover all the resources associated with realization and prefetching and to set to `NULL` all references to your `Player` so the garbage collector can dispose of it. (You do want to dispose of it, since a closed `Player` cannot be reused!)

In playing media content, it is often useful to work with one or more `Control` objects that allow you to control media processing. They are obtained from an implementer of the `Controllable` interface, in most cases a `Player`, using one of the following methods:

```
Control getControl(String controlType);
Control[] getControls();
```

A media player of a given type may support a variety of controls. The string passed in determines the name of the interface implemented by the returned control, which is typically one of the pre-defined types in the `javax.microedition.media.control` subpackage:

- `FramePositioningControl`
- `GUIControl`
- `MetaDataControl`
- `MIDIControl`
- `PitchControl`
- `RateControl`
- `TempoControl`
- `RecordControl`
- `StopTimeControl`
- `ToneControl`
- `VideoControl`
- `VolumeControl`.

If the type of control you want is not available, a NULL value is returned (which you should always check for). You will also need to cast the control appropriately before using it:

```
VolumeControl volC = (VolumeControl) player.getControl("VolumeControl");
if (volC != null)
  volC.setVolume(50);
```

The availability of support for controls depends on a number of factors, such as the media type and the phone model. Only ToneControl and VolumeControl are available as part of the MIDP audio subset. The remainder are specific to MMAPI. You can check which controls are supported by a certain Player by calling its getControls() method, which returns an array containing all available controls. You can then use instanceof to ascertain whether the control you want is available:

```
Control[] controls = player.getControls();
for (int i = 0; i < controls.length; i++) {
  if (controls[i] instanceof VolumeControl) {
    VolumeControl volC = (VolumeControl) controls[i]
    volC.setVolume(50);
  }
  if (controls[i] instanceof VideoControl) {
    VideoControl vidC = (VideoControl) controls[i]
    vidC.setDisplayFullScreen(true);
  }
}
// allow controls to be garbage collected
controls = null;
```

Note that getControl() and getControls() cannot be invoked on a Player in the UNREALIZED or CLOSED states; doing so will cause an IllegalStateException to be thrown.

Aside from using a Player, there is a further option to play simple tones or tone sequences directly using the static Manager.playTone() method. However, you normally want the additional flexibility provided by working with a Player (configured for tones) and a ToneControl (see Section 2.9.8).

The PlayerListener interface provides a playerUpdate() method for receiving asynchronous events from a Player. Any user-defined class may implement this interface and then register the PlayerListener using the addPlayerListener() method. The PlayerListener listens for a range of standard pre-defined events including STARTED, STOPPED and END_OF_MEDIA. For a list of all the standard events refer to the MMAPI specification.

2.9.4 Working with Audio Content

In this section, we demonstrate how to play audio files with code from a simple Audio Player MIDlet. We focus on the actual MMAPI code used to play the audio file.

```
// code abbreviated for clarity
public void run(){
  try {
    player = Manager.createPlayer(url);
    player.addPlayerListener(controller);
    player.realize();
    player.prefetch();
    volumeControl = (VolumeControl)player.getControl("VolumeControl");

    if (volumeControl != null) {
       volumeControl.setLevel(50);
    }
    else {
       controller.removeVolumeControl();
    }
    startPlayer();
  }
  catch (IOException ioe) {
    // catch exception connecting to the resource
  }
  catch (MediaException me) {
    // unable to create player for given MIME type
  }
}
```

A `Player` is created and a `PlayerListener` is registered with it. The controller reference serves two purposes: to facilitate callbacks to the UI indicating the progress of the initialization (this has been omitted for clarity); and to act as the `PlayerListener` which will be notified of `Player` events. The `Player` is then moved through its states. In this example, we obtain a `VolumeControl` (if the implementation supports this feature) although it is not essential for simple audio playback. The volume range provided is from 0–100. Here we set the volume level midway and start the `Player` so the audio file content can be heard from the phone's speakers.

Closing the player is a straightforward task:

```
public void closePlayer(){
  if (player != null){
    player.close();
  }
  player = null;
  volumeControl = null;
}
```

Now let us consider the `playerUpdate()` method mandated by the `PlayerListener` interface:

```
public void playerUpdate(Player p, String event, Object eventData) {
  if (event == PlayerListener.STARTED) {
    // react to the Player being started
  }
  else if (event == PlayerListener.END_OF_MEDIA){
    // react to reaching the end of media.
  }
  else if (event == PlayerListener.VOLUME_CHANGED) {
    // react to volume being changed
  }
}
```

In this example, three types of `PlayerListener` event are processed: STARTED, END_OF_MEDIA and VOLUME_CHANGED. For a complete list of events supported by `PlayerListener`, please check its documentation.

The full source code and JAR and JAD files for the Audio Player MIDlet can be downloaded from this book's website.

2.9.5 Working with Video Content

We now illustrate how to play a video with code highlights taken from a simple Video Player MIDlet (see Figure 2.21). The architecture of the Video Player MIDlet is very similar to that of the Audio Player. The

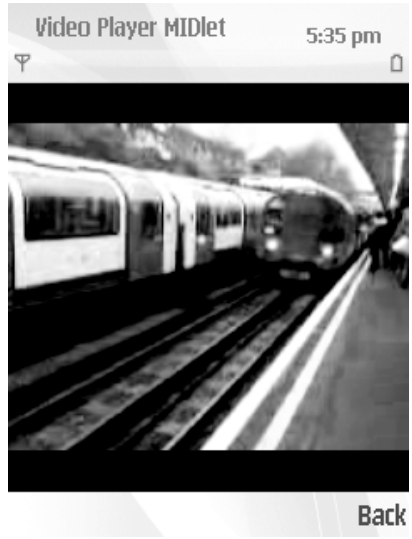

Figure 2.21 Video Player MIDlet running on a Nokia S60 device

`VideoCanvas` renders the video playback and the other classes fulfill very similar roles to their equivalents in the Audio Player MIDlet.

The resource file name is tested to ascertain its format (MPEG for the WTK emulator and 3GPP for real phones) and the appropriate MIME type. A new thread is then launched to perform the essential initialization required to play the video content. The `run()` method, mandated by the `Runnable` interface, contains the initialization of the `Player`:

```
public void run(){
  try {
    InputStream in = getClass().getResourceAsStream("/"+ resource);
    player = Manager.createPlayer(in, mimeType);
    player.addPlayerListener(controller);
    player.realize();
    player.prefetch();
    videoControl = (VideoControl)player.getControl("VideoControl");
    if (videoControl != null) {
      videoControl.initDisplayMode(
      videoControl.USE_DIRECT_VIDEO, canvas);
      int cHeight = canvas.getHeight();
      int cWidth = canvas.getWidth();
      videoControl.setDisplaySize(cWidth, cHeight);
      videoControl.setVisible(true);
      startPlayer();
    }
    else {
      controller.showAlert("Error!", "Unable to get Video Control");
      closePlayer();
    }
  }
  catch (IOException ioe) {
    controller.showAlert("Unable to access resource", ioe.getMessage());
    closePlayer();
  }
  catch (MediaException me) {
    controller.showAlert("Unable to create player",
    me.getMessage());
    closePlayer();
  }
}
```

An `InputStream` is obtained from the resource file and is used to create the `Player` instance. A `PlayerListener` (the controller) is registered with the `Player` in order to receive callbacks. The `prefetch()` and `realize()` methods are then called on the `Player` instance. Once the player is in the PREFETCHED state, we are ready to render the video content. First we must obtain a `VideoControl` by calling `getControl()` on the `Player` and casting it down appropriately.

The `initDisplayMode()` method is used to initialize the video mode that determines how the video is displayed. This method takes an integer mode value as its first argument with one of two predefined values: USE_GUI_PRIMITIVE or USE_DIRECT_VIDEO. In the case of MIDP

implementations (supporting the LCDUI), USE_GUI_PRIMITIVE results in an instance of a `javax.microedition.lcdui.Item` being returned, and it can be added to a `Form` in the same way as any other `Item` subclass. USE_DIRECT_VIDEO mode can only be used with implementations that support the LCDUI (such as Symbian OS) and a second argument of type `javax.microedition.lcdui.Canvas` (or a subclass) must be supplied. This is the approach adopted in the example code above. Methods of `VideoControl` can be used to manipulate the size and the location of the video with respect to the canvas where it is displayed. Since we are using direct video as the display mode, it is necessary to call `setVisible(true)` in order for the video to be displayed. Finally, we start the rendering of the video with the `startPlayer()` method.

The other methods of the `VideoPlayer` class are the same as their namesakes in the `AudioPlayer` class of the Audio Player MIDlet.

The `VideoCanvas` class, where the video is shown, is very simple, as the MMAPI implementation takes care of rendering the video file correctly on the canvas:

```
public class VideoCanvas extends Canvas{
  // code omitted for brevity

  // Paints background color
  public void paint(Graphics g){
    g.setColor(128, 128, 128);
    g.fillRect(0, 0, getWidth(), getHeight());
  }
}
```

The important point to note is that the `paint()` method plays no part in rendering the video. This is performed directly by the `VideoControl`. The full source code and JAR and JAD files for the Video Player MIDlet can be downloaded from this book's website.

2.9.6 Capturing Images

`VideoControl` is also used to capture images from a camera. In this case, rather than specifying a file (and MIME type) as the data source, we specify `capture://video`. Other than that, the setting up of the video player and control proceeds pretty much as in the Video Player MIDlet in Section 2.9.5.

The following code, which performs the necessary initialization of a video player and a control, is reproduced from the `VideoPlayer` class in the Video Player MIDlet example:

```
// Creates a VideoPlayer and gets an associated VideoControl
public void createPlayer() throws ApplicationException {
  try {
    player = Manager.createPlayer("capture://video");
    player.realize();
    // Sets VideoControl to the current display.
    videoControl = (VideoControl)(player.getControl("VideoControl"));
    if (videoControl == null) {
      discardPlayer();
    }
    else {
      videoControl.initDisplayMode(VideoControl.USE_DIRECT_VIDEO, canvas);
      int cWidth = Canvas.getWidth();
      int cHeight = Canvas.getHeight();
      int dWidth = 160;
      int dHeight = 120;
      videoControl.setDisplaySize(dWidth, dHeight);
      videoControl.setDisplayLocation((cWidth - dWidth)/2,
                                      (cHeight - dHeight)/2);
```

By setting the canvas to be the current one in the display, we can use it as a viewfinder for the camera. When we are ready to take a picture, we simply call `getSnapshot(null)` on the `VideoControl`:

```
public byte[] takeSnapshot() throws ApplicationException {
  byte[] pngImage = null;
  if (videoControl == null) {
    throw new ApplicationException
                          ("Unable to capture photo: VideoControl null");
  }
  try {
    pngImage = videoControl.getSnapshot(null);
  }
  catch(MediaException me) {
    throw new ApplicationException("Unable to capture photo",me);
  }
  return pngImage;
}
```

It should be noted that, if a security policy is in operation, user permission may be requested through an intermediate dialog, which may interfere with the photography!

2.9.7 Generating Tones

MMAPI also supports tone generation. Generating a single tone is simply achieved using the following method of the `Manager` class:

```
public static void playTone(int note, int duration, int volume)
                                          throws MediaException
```

The note is passed as an integer value in the range 0–127; `ToneControl.C4 = 60` represents middle C. Adding or subtracting 1 increases or lowers the pitch by a semitone. The duration is specified in milliseconds and the volume is an integer value on the scale 0–100.

To play a sequence of tones it is more appropriate to create a `Player` and use it to obtain a `ToneControl`:

```
byte[] toneSequence = { ToneControl.C4, ToneControl.C4 + 2,
                                    ToneControl.c4 + 4, ...};
try{
  Player player = Manager.createPlayer(Manager.TONE_DEVICE_LOCATOR);
  player.realize();
  ToneControl control = (ToneControl)player.getControl("ToneControl");
  control.setSequence(toneSequence);
  player.start();
}
catch (IOException ioe) { }
catch (MediaException me) { // handle }
```

A tone sequence is specified as a list of tone–duration pairs and user-defined sequence blocks, using ABNF syntax (refer to the MMAPI specification for more detail). The list is packaged as a byte array and passed to the `ToneControl` using the `setSequence()` method. To play the sequence, we simply invoke the `start()` method of the `Player`. A more sophisticated example can be found in the documentation of `ToneControl` in the MMAPI specification.

2.9.8 MMAPI on Symbian OS Phones

If you know which of the Symbian OS platforms you are targeting with a MIDlet, you can craft your code to conform to the cited capabilities. However, in practice it is more likely that you will want to write portable code which can run on several or all of the platforms, or indeed on non-Symbian OS phones with MMAPI capability. In this case, your applications need to be able to work out the supported capabilities dynamically and make use of what is available, or fail gracefully (for example, by removing certain options from menus), if the capability you want is just not available.

2.9.9 MMAPI and the MIDP Security Model

For reasons of privacy, the following Mobile Media API calls are restricted under the MIDP security model (see the *Mobile Media API Specification 1.1 Maintenance Release* at `jcp.org`):

- `RecordControl.setRecordLocation(String locator)`

- `RecordControl.setRecordStream(OutputStream stream)`
- `VideoControl.getSnapshot(String type).`

A signed MIDlet suite which contains MIDlets that make use of these APIs must explicitly request the appropriate permission in the JAD file or manifest:

```
MIDlet-Permissions: javax.microedition.media.control.RecordControl, ...
```

or:

```
MIDlet-Permissions:
          javax.microedition.media.control.VideoControl. getSnapshot, ...
```

These protected APIs are part of the Multimedia Recording function group as defined by the *Recommended Security Policy for GSM/UMTS Compliant Devices* addendum to the MIDP specification. It must also be remembered that if a MIDlet in a signed MIDlet suite makes use of a protected API of the `javax.microedition.io` package, for instance to fetch media content over HTTP, then explicit permission to access that API must be requested, even if it is fetched implicitly, perhaps by calling:

```
Manager.createPlayer("http://www.myserver.com/video.3gp")
```

Whether MIDlets in an untrusted MIDlet suite can use the protected APIs of the MMAPI depends on the security policy relating to the untrusted domain in force on the device. Under the JTWI *Security Policy for GSM/UMTS Compliant Devices*, MIDlets in an untrusted MIDlet suite can access the Multimedia Recording function group APIs with explicit permission from the user. The default user permission setting is `oneshot` (ask every time).

2.10 Wireless Messaging API

The Wireless Messaging API (JSR-120) is an optional API targeted at devices supporting the Generic Connection Framework defined in the CLDC. The Wireless Messaging API (WMA) specification defines APIs for sending and receiving SMS messages and receiving CBS messages. At the time of writing, the current release of the Wireless Messaging API is version 1.1. This contains minor modifications to the 1.0 specification to enable the API to be compatible with MIDP 2.0.

The WMA is a compact API containing just two packages:

- `javax.microedition.io` contains the platform network interfaces modified for use on platforms supporting wireless messaging connection, in particular an implementation of the `Connector` class for creating new `MessageConnection` objects.
- `javax.wireless.messaging` defines APIs which allow applications to send and receive wireless messages. It defines a base interface, `Message`, from which `BinaryMessage` and `TextMessage` both derive. It also defines a `MessageConnection` interface, which provides the basic functionality for sending and receiving messages, and a `MessageListener` interface for listening to incoming messages.

In this section, we consider sending and receiving SMS messages. We then go on to show how to use the Push Registry API to register an incoming SMS connection with a MIDlet.

2.10.1 Sending Messages

Sending an SMS message using the WMA could not be simpler, as the code paragraph below shows:

```
String address = "sms://+447111222333";
MessageConnection smsconn = null;
try {
  smsconn = (MessageConnection)Connector.open(address);
  TextMessage txtMessage = (TextMessage)
                      smsconn.newMessage(MessageConnection.TEXT_MESSAGE);
  txtmessage.setPayloadText("Hello World");
  smsconn.send(txtMessage);
  smsconn.close();
  }
  catch (Exception e) {
    // handle
  }
```

The URL address syntax for a client-mode connection has the following possible formats:

```
sms://+447111222333
sms://+447111222333:1234
```

The first format above is used to open a connection for sending a normal SMS message, which will be received in an inbox. The second format is used to open a connection to send an SMS message to a Java application listening on the specified port.

2.10.2 Receiving Messages

Receiving a message is, again, straightforward:

```
MessageConnection smsconn = null;
Message msg = null;
String receivedMessage = null;
String senderAddress = null;
try {
  conn = (MessageConnection) Connector.open(("sms://:1234");
  msg = smsconn.receive();
  ...
  // get sender's address for replying
  senderAddress = msg.getAddress();
  if (msg instanceof TextMessage) {
    // extract text message
    receivedMessage = ((TextMessage)msg).getPayloadText();
    // do something with message
    ...
  }
}
catch (IOException ioe) {
  ioe.printStackTrace();
}
```

We open a server mode `MessageConnection` by passing in a URL of the following syntax:

```
sms://:1234
```

We retrieve the message by invoking the following method on the `MessageConnection` instance.

```
public Message receive()
```

The address of the message sender can be obtained using the following method of the `Message` interface:

```
public String getAddress()
```

A server mode connection can be used to reply to incoming messages, by making use of the `setAddress()` method of the `Message` interface. In the case of a text message, we cast the `Message` object appropriately and then retrieve its contents with the `TextMessage` interface, using the method below:

```
public String getPayloadText()
```

If the message is an instance of `BinaryMessage`, then the corresponding `getPayloadData()` method returns a byte array. In practice, of course, we need the receiving application to listen for incoming messages and invoke the `receive()` method upon receipt. We achieve this by implementing a `MessageListener` interface for notification of incoming messages. The `MessageListener` mandates one method, which is called on registered listeners by the system when an incoming message arrives:

```
public void notifyIncomingMessage(MessageConnection conn)
```

The `MessageConnection` interface supplies the following method to register a listener:

```
public void setMessageListener(MessageListener l)
```

For more details on the use of the `MessageListener` interface, please download the source code, JAD and JAR files for the `SMSChat` MIDlet from this book's website.

2.10.3 WMA and the Push Registry

When implemented in conjunction with MIDP 2.0, the Wireless Messaging API can take advantage of the push registry technology. A MIDlet suite lists the server connections it wishes to register in its JAD file, or manifest, by specifying the protocol and port for the connection end point. The entry has the following format:

```
MIDlet-Push-<n>: <ConnectionURL>, <MIDletClassName>, <AllowedSender>
```

In this example, the entry in the JAD file would be as follows:

```
MIDlet-Push-1: sms://:1234, SMSMIDlet, *
```

The `<AllowedSender>` field acts as a filter indicating that the AMS should only respond to incoming connections from a specific sender. For the SMS protocol, the `<AllowedSender>` entry is the phone number of the required sender (note the sender port number is not included in the filter). Here the wildcard character '*' indicates 'respond to any sender'. The AMS responds to an incoming SMS directed to the specified `MessageConnection` by launching the corresponding MIDlet

(assuming it is not already running). The MIDlet should then respond by immediately handling the incoming message in the `startApp()` method. As before, the message should be processed in a separate thread to avoid conflicts between blocking I/O operations and normal user interaction events.

2.10.4 WMA and the MIDP Security Model

A signed MIDlet suite that contains MIDlets which open and use SMS connections must explicitly request the following permissions:

- `javax.microedition.io.Connector.sms` – needed to open an SMS connection
- `javax.wireless.messaging.sms.send` – needed to send an SMS
- `javax.wireless.messaging.sms.receive` – needed to receive an SMS

```
MIDlet-Permissions: javax.microedition.io.Connector.sms,
                    javax.wireless.messaging.sms.send
```

or:

```
MIDlet-Permissions: javax.microedition.io.Connector.sms,
                    javax.wireless.messaging.sms.send,
                    javax.wireless.messaging.sms.receive
```

If the protection domain to which the signed MIDlet suite would be bound grants, or potentially grants, the requested permissions, the MIDlet suite can be installed and the MIDlets it contains can open SMS connections and send and receive SMS messages. This can be done automatically or with explicit user permission, depending upon the security policy in effect.

Whether MIDlets in untrusted MIDlet suites can access the WMA depends on the security policy relating to the untrusted domain in force on the device. In line with the *Recommended Security Policy for GSM/UMTS Compliant Devices* addendum to the MIDP specification and the JTWI *Security Policy for GSM/UMTS Compliant Devices*, a messaging function group permission of `oneshot` requires explicit user permission to send an SMS message, but allows blanket permission (permission is granted until the MIDlet suite is uninstalled or the user changes the function group permission) to receive SMS messages.

2.10.5 WMA on Symbian OS Phones

Most Symbian OS phones in the marketplace today, even older ones such as the Nokia 3650, implement Wireless Messaging API (JSR-120). WMA has a new version (defined in JSR-205) available on most modern devices as well; its main benefit is to add Multimedia Messaging System (MMS) capabilities to the previous version of WMA. However, we won't mention it here further since only JSR-120 is mandatory on JTWI-compliant devices. WMA is a mandatory part of another umbrella specification, Mobile Service Architecture (JSR-248), which is covered in Chapter 6.

2.11 Symbian OS Java ME Certification

Majinate (***www.majinate.com***) is a company from the UK which specializes in building and testing competence in application development for Symbian OS. It operates the Accredited Symbian Developer (ASD) program on behalf of Symbian and the Accredited S60 Developer program on behalf of Nokia.

In addition to the certification exams for native application developers, Majinate also offers a test for Java ME developers working on the Symbian OS platform. The Symbian OS: J2ME exam covers the Java language and MIDP application programming concepts. It focuses on features supported by Symbian OS implementations, with the exam testing the ability of a developer to address the differences between Symbian and other Java platforms. Some of the main topics covered are:

- Java language basics
- object-orientation concepts
- package `java.lang`
- MIDlet deployment and MIDP 2 security model
- MIDlet class and lifecycle
- generic connection framework and networking
- LCDUI GUI applications
- RMS and utilities
- Java ME optional packages (Wireless Messaging and Mobile Media APIs)

This book and the standard sources (the specifications and their clarifications) are the key references for this exam. These sources have guided the creation of the curriculum and database of test questions.

For more details on the Symbian OS J2ME exam, see ***www.majinate.com/j2me.php***.

2.12 Summary

Most, if not all, Symbian OS devices currently selling in Western markets support MIDP plus a wide range of optional APIs from the Java ME JSRs, such as:

- Wireless Messaging API (JSR-120)
- Bluetooth API (JSR-82)
- PIM and FileConnection API (JSR-75)
- Mobile Media API (JSR-135)
- Mobile 3D Graphics (JSR-184).

JSR-120 and JSR-135 have been discussed here. The others are discussed in the following chapters. The latest generation of Symbian OS phones, such as Nokia N96 and Sony Ericsson G900, supports MIDP and the Mobile Service Architecture (JSR-248) APIs.

This is a very different (and much more exciting) picture from the one we found in 2004, when only two models, Nokia 6600 and Sony Ericsson P900, supported the MIDP specification. The large installed base of devices that are enabled by MIDP 2.0+ running on Symbian OS these days allows developers to create and distribute sophisticated applications using Java ME APIs to a wide target market.

We have covered the basic operations such as building and packaging MIDlets using the WTK. We have also had a good look at the APIs and components (such as LCDUI, RMS and the Game API) of MIDP, giving you the baseline information that is a prerequisite for reading the rest of the book, which assumes a certain level of knowledge of Java programming and specifically Java ME.

Lastly, we have covered the relation between MIDP and the JTWI specifications and seen how these two work hand in hand to reduce API fragmentation and provide a solid and consistent environment for developing applications with multimedia and messaging features, thus reducing the need to maintain multiple versions of your application.