

J2ME™

IN A NUTSHELL

A Desktop Quick Reference

O'REILLY®

Kim Topley

FIFTH THIRD EX1059
Page 1 of 49



J2ME™

IN A NUTSHELL

A Desktop Quick Reference

Related Titles from O'Reilly

Exploring Java™

Java™ Threads

Java™ Network Programming

Java™ Virtual Machine

Java™ AWT Reference

Java™ Language Reference

Java™ Fundamental Classes Reference

Database Programming with JDBC™ and Java™

Java™ Distributed Computing

Developing Java Beans™

Java™ Security

Java™ Cryptography

Java™ Swing

Java™ Servlet Programming

Java™ I/O

Java™ 2D Graphics

Enterprise Java Beans™

Also from O'Reilly

Java™ in a Nutshell

Java™ in a Nutshell, Deluxe Edition

Java™ Examples in a Nutshell

Java™ Power Reference: A Complete Searchable Resource on CD-ROM



J2ME™ IN A NUTSHELL

A Desktop Quick Reference

Kim Topley

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

J2ME™ in a Nutshell

by Kim Topley

Copyright © 2002 O'Reilly & Associates, Inc. All rights reserved.
Printed in the United States of America.

Published by O'Reilly & Associates, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly & Associates books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com.*

Editor: Robert Eckstein

Production Editor: Leanne Clarke Soylemez

Cover Designer: Ellie Volckhausen

Printing History:

March 2002: First Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly & Associates, Inc. The association of the image of a galago and the topic of J2ME is a trademark of O'Reilly & Associates, Inc. Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries. O'Reilly & Associates, Inc., is independent of Sun Microsystems.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly & Associates, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and the author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

0-596-00253-X
[M]

Table of Contents

Preface ..... *ix*

Part I: Introduction to the Java 2 Micro Edition Platform API

Chapter 1—Introduction *3*

What Is the J2ME Platform? 3
J2ME Specifications 8
J2ME and Other Java Platforms 9

Chapter 2—The Connected Limited Device Configuration *11*

The CLDC Java Virtual Machine 11
The CLDC Class Libraries 19
KVM Debugging 28
Advanced KVM Topics 33

Chapter 3—The Mobile Information Device Profile and MIDlets *44*

MIDP Overview 44
The MIDP Java Platform 48
MIDlets and MIDlet Suites 49
MIDlet Execution Environment and Lifecycle 56
Developing MIDlets 60
Delivery and Installation of MIDlets 75

FIFTH THIRD EX1059
Page 6 of 49

Chapter 4—MIDlet User Interfaces *84*

User Interface Overview 85
The High-Level User Interface API 89

Chapter 5—The Low-Level MIDlet User Interface API *140*

The Canvas Class 140
Painting and the Graphics Class 144
Graphics Attributes 145
Drawing Lines and Arcs 148
Translating the Graphics Origin 155
A Simple Animation MIDlet 156
The Graphics Clip 160
Rendering Text 163
Images 168
Event Handling 173
Multithreading and the User Interface 178

Chapter 6—Wireless Java: Networking and Persistent Storage *181*

A Networking Architecture for Small Devices 182
Sockets 185
Datagrams 190
HTTP Connections 195
Persistent Storage 210

Chapter 7—The Connected Device Configuration and Its Profiles *230*

The CDC 230

Chapter 8—J2ME Command-Line Tools *242*

cvm: The Connected Device Configuration Virtual Machine 242
kdp: The KVM Debug Proxy 247
kvm: The Kilobyte Virtual Machine 249
midp: The MID Profile Execution Environment 252
emulator: The J2ME Wireless Toolkit Emulator 256
preverify: The KVM Class Preverifier 259
MakeMIDPApp: JAD to PRC Conversion Tool 262
MEKeyTool: Public Key Certificate Management Tool 264

FIFTH THIRD EX1059
Page 7 of 49

Chapter 9—J2ME Programming Environments *267*

The J2ME Wireless Toolkit 268
MIDP for PalmOS 282
J2ME and Forte For Java 292
Other Integrated Development Environments 298

Part II: API Quick Reference

How to Use This Quick Reference *303*

Chapter 10—J2ME Packages and Classes *312*

J2ME Packages 312
J2SE Packages Not Present in J2ME 313
J2ME Package Contents 314

Chapter 11—The java.io Package *338*

Chapter 12—The java.lang Package *351*

Chapter 13—The java.util Package *374*

Chapter 14—The javax.microedition.io Package *385*

Chapter 15—The javax.microedition.lcdui Package *396*

Chapter 16—The javax.microedition.midlet Package *422*

Chapter 17—The javax.microedition.rms Package *425*

Chapter 18—Class, Method, and Field Index *434*

Index *443*

FIFTH THIRD EX1059
Page 8 of 49



CHAPTER 3

The Mobile Information Device Profile and MIDlets

The Connected Limited Device Configuration provides the basis for running Java on devices that have insufficient resources to support a complete virtual machine together with a full version of the J2SE core packages. However, if you are an application developer, it is extremely unlikely that you will ever need to write software that works solely with the APIs provided by CLDC, because it contains nothing that allows for interaction with users, storage devices, or a network. CLDC is intended to be a base layer on top of which a range of profiles that provide the missing facilities can be provided, in a form suitable for the class of device for which each profile is designed. The *Mobile Information Device Profile*, or MIDP for short, is one such profile, intended for use on small footprint devices with a limited user interface in the form of a small screen with some kind of input capability. This chapter introduces MIDP; in the following two chapters, we'll look in more detail at how it supports user interfaces, networking, and persistent storage of information.

MIDP Overview

MIDP is a version of the Java platform based on CLDC and KVM that is aimed at small footprint devices, principally cell phones and two-way pagers. It is also suitable for running on PDAs, and an implementation is available for PalmOS Version 3.5 and higher. (In the longer term, it is intended that these devices use the PDA profile, which is also hosted by CLDC.) The MIDP specification was developed under the Java Community Process and is available for download from *http://jcp.org/jsr/detail/37.jsp*.

The logical position of MIDP within the software architecture of a device that implements it is shown in Figure 3-1. The software that implements MIDP runs in the KVM supplied by CLDC and provides additional services for the benefit of application code written using MIDP APIs. MIDP applications are called *MIDlets*. As Figure 3-1 shows, MIDlets can directly use both MIDP facilities and the APIs described in Chapter 2 that MIDP inherits from CLDC itself. MIDlets do not access

FIFTH THIRD EX1059
Page 9 of 49

the host platform's underlying operating system and cannot do so without becoming nonportable. Because the KVM does not support JNI, the only way for a MIDP application to access native platform facilities directly is by linking native code into a customized version of the virtual machine.

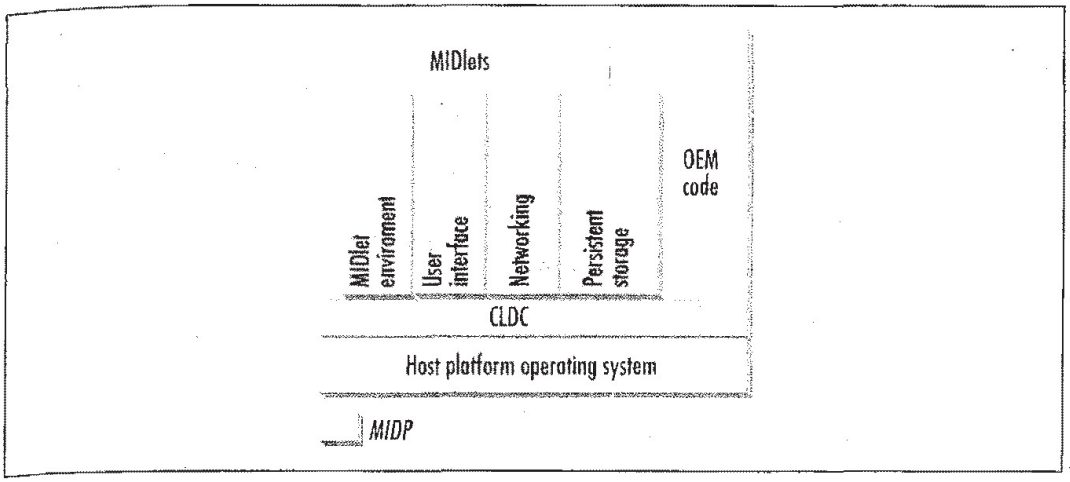


Figure 3-1: The Mobile Information Device Profile

Sun provides a reference implementation of MIDP that can be used on Windows; the Wireless Toolkit, which contains versions of MIDP for Windows, Solaris and Linux; and a separate MIDP product for use on PalmOS-based PDAs. Device manufacturers typically use the Sun reference implementation as the basis for their own products. They usually integrate additional code as part of their MIDP implementation to provide management features such as installation, removal, and management of MIDlets that are not portable between devices and hence not part of the reference software. As shown in Figure 3-1, this code (labeled "OEM Code") may use some combination of MIDP and CLDC services and will also depend on the host platform's operating system. Some parts of the core MIDP software are themselves device-dependent and thus are also supplied by the manufacturer. These typically include parts of the networking support, the user interface components, and the code that provides persistent storage.

MIDP Hardware Requirements

As mentioned earlier, MIDP is intended for small devices with limited memory, CPU, and display capabilities. The minimum hardware requirements are described in the following sections.

Memory

As you'll see over the next few chapters, MIDP includes quite a lot of software that is not part of the core Java platform and therefore requires more memory than the minimal CLDC environment is obliged to supply. The MIDP specification requires at least 128 KB of RAM be available to store the MIDP implementation itself, over and above whatever is needed by CLDC. In addition to this, there must be at least 32 KB available for the Java heap. In practice, a 32 KB heap is very limiting and demands that the developer exercise great care when allocating objects and take

FIFTH THIRD EX1059
Page 10 of 49

all possible steps to avoid holding references to objects longer than necessary, in order to allow the garbage collector to reclaim heap space as quickly as possible. As well as the RAM requirement, MIDP devices must also supply at least 8 KB of nonvolatile memory to be used as persistent storage so that MIDlets can save information in such a way that it is not lost when the device is switched off. The content of this storage is not guaranteed to be preserved over battery changes, however, and there is a general expectation that the device also provides some way (such as the PDA "hot sync" mechanism) to back up its content to a more permanent location.

Display

MIDP devices are characterized by small displays. The specification requires that the screen should be at least 96 pixels wide and 54 pixels high and that each pixel be (approximately) square. The screen must support at least two colors, and many cell phones are capable of no more than this. At the top of the range, PDAs typically have screens with 160 pixels in each direction and support as many as 65,536 different colors. This wide disparity in capability provides the developer who wants to write a fully portable MIDlet with some interesting challenges, and it has led to some trade-offs in the MIDP user interface library, as we'll see in Chapters 4 and 5.

Input device

As with displays, there are several different types of input device that might be found on a MIDP platform. At one end of the spectrum, the more sophisticated devices, like the RIM wireless handheld, have a complete alphanumeric keyboard, as shown on the left of Figure 3-2. Similarly, PalmOS-based handhelds allow the user to "write" on a special area of the screen using a form of shorthand known as Graffiti; they also provide a simulated onscreen keyboard for users who prefer a more traditional approach. The screenshot on the right side of Figure 3-2 shows the Graffiti area of a Palm handheld.

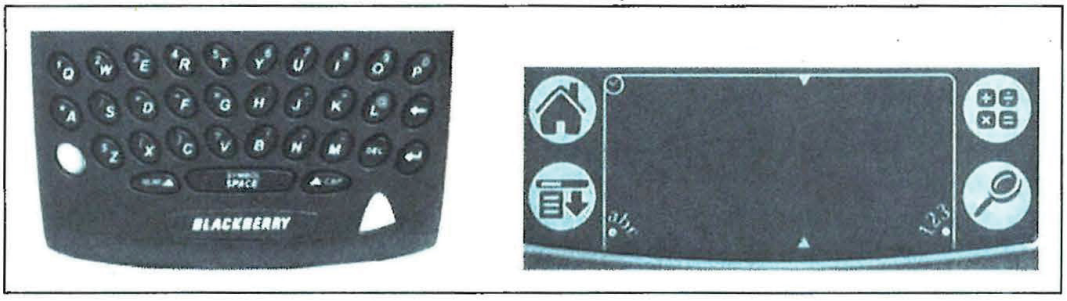


Figure 3-2: Handheld input devices

Contrast these highly functional keyboards (or keyboard substitutes) with the more basic one that you'll find on most cell phones, an example of which is shown in Figure 3-3. Keyboards like this provide relatively easy numeric input, but they require slightly more work on the part of the user to type alphabetic characters, and there are almost no special characters available.

FIFTH THIRD EX1059
Page 11 of 49

The minimum assumption made by the MIDP specification is that the device has the equivalent of a keypad that allows the user to type the numbers 0 through 9, together with the equivalent of arrow keys and a select button as shown in the diamond-shaped arrangement at the top of Figure 3-3, where the select button is the white circle between the arrows. These requirements are directly met by cell phones and may be satisfied in various ways on other devices. On the Palm, for example, there are buttons that may be programmed to act as directional arrows, while the select operation can be performed by tapping the screen with the stylus. As we'll see in Chapter 5, this cut-down representation of the input device is reflected in the APIs that handle the user interface, and it requires the developer to be careful when handling events from whatever passes for the keyboard on the device on which a MIDlet is running.



Figure 3-3: A typical cell phone keypad

Connectivity

Mobile information devices have some kind of network access, whether it's the built-in wireless connection in a cell phone or pager, or a separate modem attached to a PDA. MIDP does not assume that devices are permanently attached to a network or that the network directly supports TCP/IP. It does, however, require that the device vendor provide at least the illusion that the device supports HTTP 1.1, either directly over an Internet protocol stack, as would be the case for a Palm handheld connected to a modem, or by bridging a wireless connection to the Internet via a WAP gateway. This provision allows developers to write network-aware MIDlets that work equally well (other than performance differences due to differing network bandwidth) across all supported platforms.

MIDP Software Requirements

Sun's reference version of MIDP is not a commercial product. Device vendors are expected to port the reference implementation to their own hardware and software by implementing code that bridges the gap between the expectations of Sun's reference code and the vendor's hardware and operating system software. As

FIFTH THIRD EX1059
Page 12 of 49

with the hardware described previously, the reference implementation makes the following assumptions about the capabilities offered by the software base on which it will be hosted (shown as "Host Platform Operating System" in Figure 3-1:

- The operating system must provide a protected execution environment in which the JVM can run. Because CLDC supports the threading capabilities of J2SE, the host platform ideally supports multithreading, and, if it does, the KVM can make direct use of it. However, MIDP implementations are required to provide the illusion of multithreading even when this is not available from the native operating system. They do this by sharing the single available thread between the Java threads that belong to application code and those used within the VM and the MIDP and core libraries.
- Networking support is required in some form. On some platforms, such as PalmOS, a socket-level API is available, over which the mandatory MIDP HTTP support can be implemented. In the case of devices that do not offer such a convenient interface, including those that do not have direct connectivity to an IP-based network, the vendor is required to provide a means for HTTP to be bridged from the device's own network to the Internet. The networking aspects of MIDP are discussed in detail in Chapter 6.
- The software must provide access to the system's keyboard or keypad (or equivalent) and a pointing device, if it is available. The software must be able to deliver events when keys are pressed and released and when the pointing device is moved or activated. (For example, for a handheld with a stylus, the software must deliver an event when the stylus touches the screen, when it is lifted off the screen, and when it moves over the screen.) The vendor is required to map whatever codes are delivered by the user's keystrokes to a standard set of values so that similar keystrokes lead to the same results across different hardware platforms. This issue is discussed further in Chapter 5.
- It must be possible to access the device's screen. MIDP allows MIDlets to treat the screen as a rectangular array of pixels, each of which may be independently set to one of the colors supported by the device. Therefore, it is required that the software provide access to the screen as if it were a bit-mapped graphics device. MIDP user interfaces and graphics are covered in detail in Chapters 4 and 5.
- The platform must provide some form of persistent storage that does not lose its state when the device is switched off (that is, when it is in its minimum power mode, but not necessarily when it has no power at all). MIDP provides record-level access to this storage and therefore requires that the host software supply some kind of programmatic interface to its persistent storage mechanism. The MIDP storage APIs are described in Chapter 6.

The MIDP Java Platform

The Java platform available to MIDlets is that provided by CLDC as described in Chapter 2, together with a collection of MIDP-specific packages arranged under the `javax.microedition` package hierarchy. The core libraries themselves are almost unaffected by the MIDP specification; the only change is the addition of the

FIFTH THIRD EX1059
Page 13 of 49

J2SE 1.3 timer facility in the `java.util` package, which will be covered in the later section "Timers and TimerTasks." The MIDP specification also places the following requirements on the core libraries:

- Like applets, MIDlets are managed in an execution environment that is slightly different from that of a Java application. As you'll see shortly, the initial entry point to a MIDlet is not the `main()` method of its MIDlet class, and the MIDlet is not allowed to cause the termination of the Java VM. In order to enforce this restriction, the `exit()` methods in both the `System` and `Runtime` classes are required to throw a `SecurityException` if they are invoked.
- In addition to the system properties defined by CLDC, MIDP devices must set the `microedition.locale` property to reflect the locale in which the device is operating. The locale names are formed in a slightly different way from those used by J2SE, because the language and country components are separated by a hyphen instead of an underscore character. A typical value for this property would be `en-US` on a MIDP device, whereas a J2SE developer would expect the locale name in the form `en_US`. Since both MIDP and CLDC provide almost no support for localization, however, the precise format of this property is of little direct interest to MIDlets. Instead, it is intended to be used when installing MIDlets from external sources, to allow the selection of a version of the MIDlet suitable for the device owner's locale. The property must therefore be properly interpreted by the agent (perhaps a servlet running in a web server) that supplies the software.
- The system property `microedition.profiles` must contain at least the value `MIDP-1.0`. In the future, as new versions of the MIDP specification are released and implemented, devices that support multiple profiles may list them all in this profile, using spaces to separate the names.

In summary, the values of the system properties that are introduced by or changed by MIDP relative to the requirements placed by CLDC, are shown in Table 3-1.

Table 3-1: MIDP System Properties

Property	*Meaning*	*Value*
`microedition.locale`	The current locale of the device	e.g., `en-US`
`microedition.profiles`	Blank-separated list of supported profiles	`MIDP-1.0`

MIDlets and MIDlet Suites

Java applications that run on MIDP devices are known as MIDlets. A MIDlet consists of at least one Java class that must be derived from the MIDP-defined abstract class `javax.microedition.midlet.MIDlet`. MIDlets run in an execution environment within the Java VM that provides a well-defined lifecycle controlled via methods of the `MIDlet` class that each MIDlet must implement. A MIDlet can also use methods in the MIDlet class to obtain services from its environment, and it must use only the APIs defined in the MIDP specification if it is to be device-portable.

FIFTH THIRD EX1059
Page 14 of 49

A group of related MIDlets may be collected into a *MIDlet suite*. All of the MIDlets in a suite are packaged and installed onto a device as a single entity, and they can be uninstalled and removed only as a group. The MIDlets in a suite share both static and runtime resources of their host environment, as follows:

- At runtime, if the device supports concurrent running of more than one MIDlet, all active MIDlets from a MIDlet suite run in the same Java VM. All MIDlets in the same suite therefore share the same instances of all Java classes and other resources loaded into the Java VM. Among other things, this means that data can be shared between MIDlets, and the usual Java synchronization primitives can be used to protect against concurrent access not only within a single MIDlet, but also between concurrently executing MIDlets from the same suite.
- Persistent storage on the device is managed at the MIDlet suite level. MIDlets can access their own persistent data and that of other MIDlets in the same suite. However, it is not possible for a MIDlet to gain access to persistent storage owned by another suite, because the naming mechanism used to identify the data implicitly includes the MIDlet suite. This is partly to avoid unintended name clashes between MIDlets obtained from unrelated sources, and partly as a security measure so that a MIDlet's data cannot be read or corrupted by malicious code imported from an unreliable source.

As an example of the sharing of classes and data between MIDlets, suppose a MIDlet suite contains a class called `Counter`, intended to keep count of the number of instances of MIDlets from that suite are running at any given time.

```
public class Counter {
    private static int instances;

    public static synchronized void increment() {
        instances++;
    }

    public static synchronized void decrement() {
        instances--;
    }

    public static int getInstances() {
        return instances;
    }
}
```

Only a single instance of this class will be loaded in the Java VM, no matter how many MIDlets from the suite that supplies it are running in that VM. This means that the same static variable `instances` is used by all of these MIDlets, and, therefore the `increment` and `decrement` methods all affect the same counter. The fact that these methods are synchronized protects the `instances` variable from concurrent access by any threads in all of the MIDlets.

MIDlet Security

For the developer, dealing with MIDlet security is a very simple issue, because there isn't any! The Java security model used in J2SE is both powerful and flex-

FIFTH THIRD EX1059
Page 15 of 49

ible, but it is expensive in terms of memory resources, and it requires a certain amount of administration that may be beyond the knowledge expected of a mobile device user. Therefore, neither CLDC nor MIDP includes any of the security checking of API calls that is available in J2SE, with the exception of the `Runtime` and `System exit()` methods, which can never be used by a MIDlet.

For the mobile device owner, this means that a MIDlet appears to be more of a potential threat than an applet would to a browser user, because the MIDlet is not constrained by the Java applet "sandbox" that the browser imposes via a `SecurityManager`. A mobile device owner needs to be careful when installing MIDlets and, preferably, he should accept software only from trusted sources. Unfortunately, at the time of writing, there is no way for the user to be completely sure who is actually providing a MIDlet or that the MIDlet code has not been interfered with en route to the device; the authentication mechanisms that provide this for the J2SE platform (i.e., public key cryptography and certificates) are not a standard part of the MIDP specification. The secure version of the HTTP protocol (HTTPS), which will help to alleviate this problem, is under consideration for inclusion in a future version of MIDP. In the meantime, there is limited security against malicious MIDlets. There are no MIDlet APIs that allow access to information already on the device, such as address and telephone number lists or calendars, and it is not possible for a MIDlet to directly control the device. As you'll see in Chapter 6, a MIDlet can store information on a device, but that storage is private to that MIDlet and its suite, so the MIDlet can harm only its own data.

MIDlet Packaging

MIDlets need to be properly packaged before they can be delivered to a device for installation. The `MIDlet` subclass that acts as the main entry point for the MIDlet, together with any other classes that it requires (apart from those provided by MIDP itself) and any images or other files to which it needs access at runtime, must be built into a single JAR file. Packaging information that tells the device what is in the JAR must be supplied in the JAR's manifest file. Similar packaging information is also provided in another file called the *Java application descriptor* (or JAD file for short), which is held separately from the JAR. A JAR may contain more than one MIDlet, in which case all the MIDlets are deemed to be in the same MIDlet suite. To put the same thing another way, all MIDlets that are in the same MIDlet suite *must* be packaged in the same JAR.

Both the manifest file and the JAD file are simple text files in which each line has the form:

```
attribute-name: attribute-value
```

The name and value are separated by a colon and optional whitespace. All the attributes that are of relevance to the installation of MIDlets have names with the prefix "MIDlet-". A complete list of these attributes, together with a short description of their associated values, can be found in Table 3-2. The values in the JAR and JAD columns indicate whether the attribute is mandatory (M), optional (O) or ignored (I) in the file corresponding to that column.

FIFTH THIRD EX1059
Page 16 of 49

Table 3-2: MIDlet Packaging Attributes

Attribute Name	*JAR*	*JAD*	*Value and Meaning*
`MIDlet-Name`	M	M	The name of the MIDlet suite packaged in the JAR file. This name may be displayed to the user.
`MIDlet-Version`	M	M	The version number of the MIDlet suite packaged in the JAR file. Version numbers take the form a.b.c (for example 1.2.3), where larger values in each field indicate a newer version, with the leftmost field taking precedence. For example, version 1.2.5 is taken to be more recent than version 1.2.3, and, similarly, version 2.1.5 is newer than 1.3.7.
`MIDlet-Vendor`	M	M	The name of the MIDlet suite provider. This is free-form text that is intended for display to the user.
`MIDlet-`*n*	M	I	Attributes that describe the MIDlet in the MIDlet suite. The value *n* is replaced by a numeric value starting from 1 to identify individual MIDlets. The format of the value associated with this attribute is described in the text.
`MicroEdition-Profile`	M	I	The version or versions of the MIDP specification that the MIDlets in this suite can work with. Where more than one version appears, they must be separated by spaces. The versions specified are compared to those listed in the `microedition.profiles` property of the target device to determine whether the MIDlets are compatible with them. `MIDP-1.0` is a typical value for this attribute.
`MicroEdition-Configuration`	M	I	The J2ME configuration required by the MIDlets in this suite. This value is compared to the target device's `microedition.configuration` property to determine compatibility.
`MIDlet-Description`	O	O	A description of the MIDlet suite intended to be displayed to the user.
`MIDlet-Icon`	O	O	An icon that may be used to represent the MIDlet suite during or following installation. The icon must be a Portable Network Graphics (PNG) file.
`MIDlet-Info-URL`	O	O	The URL of a file that contains further information describing the MIDlet suite. The content of this file may be displayed to the user to allow the user to decide whether to install the MIDlet suite.
`MIDlet-Data-Size`	O	O	The minimum amount of persistent storage that this MIDlet suite requires. This refers to space used for the long-term storage of data used by the MIDlet suite, not the space required to install and manage the MIDlet suite itself. It is specified in bytes. If this attribute is not supplied, it is assumed that the MIDlet suite does not require persistent storage. Whether or not MIDlets can use more persistent storage space than they initially request is device-dependent.

FIFTH THIRD EX1059
Page 17 of 49

Table 3-2: MIDlet Packaging Attributes (continued)

Attribute Name	*JAR*	*JAD*	*Value and Meaning*
`MIDlet-Jar-URL`	I	M	The URL of the JAR file that contains the MIDlet or MIDlet suite described by these attributes. This attribute is used only in the application descriptor.
`MIDlet-Jar-Size`	I	M	The size of the MIDlet JAR file in bytes. This attribute is used only in the application descriptor.
`MIDlet-Install-Notify`	I	O	A URL used to report the success or failure of MIDlet installation performed from a remote server. This attribute is not included in the current MIDP specification, but it is supported by the Wireless Toolkit. See the later section "Delivery and Installation of MIDlets" for further details.
`MIDlet-Delete-Confirm`	I	O	A message to be displayed to the user before the MIDlets are deleted from the device on which they are installed. Like `MIDlet-Install-Notify`, this attribute is not currently included in the formal specification.
MIDlet-specific attributes	O	O	MIDlet developers can provide limited configurability for MIDlets by including attributes that can be retrieved at runtime.

As you can see, many of the attributes must be supplied in both the manifest file, which resides in the JAR, and in the JAD file, which does not. To see why, it is necessary to understand why two files are used.

The job of the manifest file is to indicate to the device the name and version of the MIDlet suite in the JAR and to specify which of the class files it contains correspond to the individual MIDlets. In order to make use of this information, however, the device must download the JAR and extract the manifest. Having done this, it can then display the values associated with the `MIDlet-Name`, `MIDlet-Version`, and `MIDlet-Vendor` attributes and the optional `MIDlet-Description` and `MIDlet-Icon` attributes. These attributes allow the user to decide whether the MIDlets should be installed. However, the JAR for a MIDlet suite might be quite large and may take some time to retrieve over the relatively slow networks to which mobile devices typically have access. If the only useful description of its content were in the JAR itself, a lot of time might be wasted transferring large files that are immediately rejected as uninteresting.

To solve this problem, some of the attributes from the manifest file, together with extra information, is duplicated in the JAD file. Instead of downloading the whole JAR, a MIDP device first fetches its JAD file, which is much smaller than the JAR and can be transferred quickly. The device then displays the JAD file's contents to the user so that she can decide whether to fetch the JAR file. The JAD contains some attributes that come from the manifest file and others that do not appear in the manifest. The common attributes are as follows:

```
MIDlet-Name
MIDlet-Vendor
MIDlet-Version
```


FIFTH THIRD EX1059
Page 18 of 49

```
MIDlet-Description
MIDlet-Icon
MIDlet-Info-URL
MIDlet-Data-Size
```

These attributes (with the possible exception of the last one) can all be presented to the user as an aid to deciding whether the content of the corresponding JAR file is interesting enough to download. The first three of these attributes are mandatory in both JAR and JAD files, and the MIDP specification requires that their values be identical. The remaining attributes are all optional. If they appear in both the manifest and the JAD file, the value in the JAD file takes precedence over that in the manifest (and at this stage, the device can see only the value in the JAD file).

The JAD file also contains two other attributes that are not present in the manifest file:

```
MIDlet-Jar-Size
MIDlet-Jar-URL
```

The `MIDlet-Jar-Size` attribute can be displayed to the user to help determine how long it will take to fetch the JAR; it also enables the user to guess whether the device has enough free space to install the JAR. Assuming the user decides to install the MIDlet suite, the next step is to fetch the JAR itself, which can be found by using the value of the `MIDlet-Jar-URL` attribute.

Suppose a company called "Wireless Java Inc." creates a suite of MIDlets called `WirelessTrader` that allow a user to do online stock trading from a MIDP device. The suite contains two MIDlets, one for trading, the other for simply browsing through stock prices. The main classes for these two MIDlets are called `com.wireless.TradeMIDlet` and `com.wireless.BrowseMIDlet`, and they make use of common code in the `com.wireless.Utils` class. The manifest for this suite would look something like this:

```
MIDlet-Name: WirelessTrader
MIDlet-Vendor: Wireless Java Inc.
MIDlet-Version: 1.0.1
MIDlet-Description: A set of MIDlets for online trading.
MIDlet-Icon: /com/wireless/icons/wireless.png
MIDlet-Info-URL: http://www.wireless.com/trader/info.html
MIDlet-Data-Size: 512
MicroEdition-Profile: MIDP-1.0
MicroEdition-Configuration: CLDC
MIDlet-1: StockTrader,/com/wireless/icons/trader.png,com.wireless.
TradeMIDlet
MIDlet-2: StockBrowser,/com/wireless/icons/browser.png,com.
wirelessBrowseMIDlet
```

In the JAR, this file would appear as `META-INF/MANIFEST.mf`. The JAR would also include the following files:

/com/wireless/BrowseMIDlet.class
/com/wireless/TradeMIDlet.class
/com/wireless/Utils.class

FIFTH THIRD EX1059
Page 19 of 49

/com/wireless/icons/browser.png
/com/wireless/icons/trader.png
/com/wireless/icons/wireless.png

Note the following about the attributes in the manifest file and the content of the JAR:

- The JAR contains the two MIDlet class files and the class file for `com.wireless.Utils`, which contains code that is used by both MIDlets. This latter file, however, does not need to be referenced from the manifest file. The JAR also contains the three icons that are referred to from the manifest file.
- The `MIDlet-Icon` attribute contains the absolute path of the icon file for the MIDlet suite, relative to the JAR file itself.
- Each MIDlet has an attribute that describes it; the attribute's name is of the form `MIDlet-n`, where `n` is an integer. The value of this attribute has the following form:

  ```
  name,icon,class
  ```

  `name` is the name of the MIDlet within the MIDlet suite. `icon` is the full path of the icon that the device may use along with the MIDlet name when displaying the content of the MIDlet suite to the user. `class` is the name of the MIDlet's main class. The icon is optional; if no icon is required, it should be omitted:

  ```
  MIDlet-2: StockBrowser,,com.wireless.BrowseMIDlet
  ```

  Note that even if an icon is specified, the device is not obliged to display it. The same applies to the MIDlet suite icon defined by the optional `MIDlet-Icon` attribute.

The JAD file for this suite can be constructed like this:

```
MIDlet-Name: WirelessTrader
MIDlet-Vendor: Wireless Java Inc.
MIDlet-Version: 1.0.1
MIDlet-Description: A set of MIDlets for online trading.
MIDlet-Info-URL: http://www.wireless.com/trader/info.html
MIDlet-Data-Size: 512
MIDlet-Jar-Size: 10312
MIDlet-Jar-URL: http://www.wireless.com/trader/Midlets.jar
```

This file contains the information that the device displays to the user, together with the URL of the MIDlet suite JAR. In this case, the common attributes have the same values in the manifest and the JAR, but it is possible to override the `MIDlet-Description`, `MIDlet-Icon`, `MIDlet-Info-URL`, and `MIDlet-Data-Size` attributes by specifying different values in the JAD file.

In order to be fully portable, the JAD file should be encoded using ISO-8859-1, because all MIDP implementations are required to support this character encoding. The successful use of any other encoding depends on the target device, which may not support the encoding, and the way in which the JAD file is transported to the device. If, for example, the file is fetched using HTTP, the Content-Type header can be used to specify the encoding as described in the later section "Delivery and Installation of MIDlets." In some cases, it is useful to be able to include in the JAD

FIFTH THIRD EX1059
Page 20 of 49

file Unicode characters that are not available in the ISO-8859-1 encoding or that are not easy to access from a standard keyboard. The MIDP reference implementation allows you to use Unicode escape sequences of the form `\uxxxx` to overcome encoding limitations. For example, the following line includes the copyright character (Unicode value 00A9) in the MIDlet suite description:

```
MIDlet-Description: A set of MIDlets for online trading. \u00A9 Wireless
Java Inc.
```

Although this feature is available in the MIDP reference implementation, it is not mentioned in the MIDP specification, so there is no guarantee that real devices will actually support it.

At runtime, a MIDlet can access files from its JAR using the `getResourceAsStream()` method of `java.lang.Class`. Any file in the JAR, apart from class files, can be accessed this way. This is typically how you would include images or text files that should be displayed in the user interface, an example of which will be shown in Chapter 4. A MIDlet can also define its own private attributes in the manifest file and the JAD and retrieve them at runtime, as you'll see in "Developing MIDlets," later in this chapter.

MIDlet Execution Environment and Lifecycle

All MIDlets are derived from the abstract base class `javax.microedition.midlet.MIDlet`, which contains methods that the MIDP platform calls to control the MIDlet's lifecycle, as well as methods that the MIDlet itself can use to request a change in its state. A MIDlet must have a public default constructor (that is, a constructor that requires no arguments), which may be one supplied by the developer if there is any initialization to perform or, when there are no explicit constructors, the empty default constructor inserted by the Java compiler. This is what a skeleton MIDlet class might look like:

```
public class MyMIDlet extends MIDlet {

    // Optional constructor
    MyMIDlet() {
    }

    protected void startApp() throws MIDletStateChangedException {
    }

    protected void pauseApp() {
    }

    protected void destroyApp(boolean unconditional)
                              throws MIDletStateChangedException {
    }
}
```

At any given time, a MIDlet is in one of three states: Paused, Active, or Destroyed. A state diagram that shows how these states are related and the legal state transitions is shown in Figure 3-4.

FIFTH THIRD EX1059
Page 21 of 49

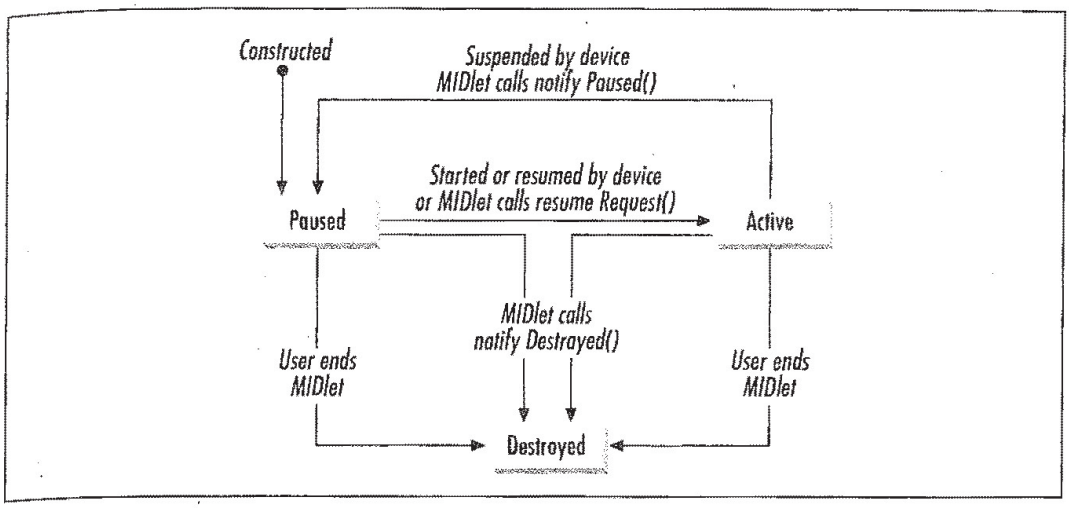


Figure 3-4: The lifecycle of a MIDlet

When a MIDlet is loaded, it is initially in the Paused state. The usual class and instance initialization is then performed—that is, static initializers are called the first time the MIDlet class is loaded, all instance initializers are invoked when the MIDlet instance is created, and its public, no-argument constructor is then invoked. If the MIDlet throws an exception during the execution of its constructor, the MIDlet is destroyed. If the MIDlet does not throw an exception, it is scheduled for execution at some later time. Its state is changed from Paused to Active, and its `startApp()` method is called. The MIDlet class declares this method as follows:

```
protected void startApp() throws MIDletStateChangeException;
```

That this method is abstract means that you must implement it in your MIDlet, and that it is protected implies that it will be called either from the MIDlet class itself or from another class in the `javax.microedition.midlet` package. In the reference implementation, the MIDlet lifecycle methods are called from a class in this package called `Scheduler`, but there is nothing in the MIDP specification that requires this class be used. Licensees may provide their own scheduler implementations, provided that it supports the MIDlet lifecycle as described in this section. It is very common for MIDlet developers to redefine the `startApp()` method as public, which is certainly a safe option, but this should not be necessary because vendor implementations must continue to work even if these methods are declared as protected.

The `startApp()` method may complete normally, in which case the MIDlet is allowed to run, or it may inform the MIDP platform that the MIDlet does not want to run at this point. There are several ways to achieve the latter:

- If the `startApp()` method detects an error condition that stops it from completing, but which might not exist later (i.e., a transient error condition), it should throw a `MIDletStateChangeException`. This moves the MIDlet back to the Paused state, so that another attempt to start it can be made later.
- If the `startApp()` method detects an error condition from which recovery is likely never to be possible (a nontransient error condition), it should call its `notifyDestroyed()` method, which is described a little later.

FIFTH THIRD EX1059
Page 22 of 49

- Finally, the MIDlet may throw an exception other than `MIDletStateChangeException`, either deliberately or because a method that it invokes throws the exception, and the `startApp()` method does not catch it. In this case, it is assumed that a fatal error has occurred, and the MIDlet is destroyed by calling its `destroyApp()` method (described later).

If the MIDlet does none of these things, it is in the Active state and will be allowed to run until it is either paused or destroyed. A MIDlet returns after completing its `startApp()` method, and it does not have a method that contains the main logic to which control could be passed, so where is the MIDlet's code placed? Usually, a MIDlet has a user interface and executes code as a result of events generated by key presses or pointer movements. MIDlets can also start separate background threads to run code that does not depend on the user interface, or they can use a timer to schedule work periodically, as will be shown later. If you take these approaches, it is important to manage the background threads and/or timers appropriately when the MIDlet itself is paused or destroyed.

At any time, the MIDP platform can put a MIDlet into the Paused state. On a cell phone, for example, this might happen when the host software detects an incoming call and needs to release the phone's display so the user can answer the call. When a MIDlet is paused, its `pauseApp()` method is called:

```
protected abstract void pauseApp();
```

As with `startApp()`, a MIDlet is required to provide an implementation for this method. The appropriate response to this state change depends on the MIDlet itself, but, in general, it should release any resources it is holding and save the current state so it can restore itself when it is reactivated later.

The main consequence of being moved to the Paused state is that the MIDlet no longer has access to the screen; any threads that it created are not automatically terminated, and timers remain active. A MIDlet may choose to terminate any open network connections or background threads and cancel active timers when told to pause, but it is not obliged to do so.

If the host platform decides to resume a paused MIDlet, because the incoming call has terminated, for example, the MIDlet's `startApp()` method is invoked again to notify the MIDlet that it has access to the screen. As a consequence, a MIDlet's `startApp()` method should be written carefully to distinguish, if necessary, between the first time that it is called, which signifies that the MIDlet is being started for the first time, and subsequent calls notifying resumption from the Paused state, to prevent resources from being allocated multiple times. Of course, if a MIDlet reacts to being moved to the Paused state by releasing all of its resources, it would probably be appropriate to execute the same initialization code in `startApp()` to reallocate the resources upon resumption. However, a properly written MIDlet would still take special action in the `startApp()` method to restore the user interface and its internal state to the way it was before it was paused, rather than show the initial screen again.

The fact that the `startApp()` method can be invoked more than once in the lifetime of a MIDlet raises the question of whether initialization should be performed here or in the MIDlet's constructor. The developer is free to choose the more convenient location to allocate resources and prepare the MIDlet's state. In

FIFTH THIRD EX1059
Page 23 of 49

general, resources that will be released in pauseApp() should be allocated in startApp(). Other resources can be allocated in either startApp() or the constructor, with care being taken to ensure that allocations performed in startApp() are not repeated following resumption from the Paused state.

An important difference between the startApp() method and the constructor is that, according to the MIDP specification, the MIDlet is guaranteed to be able to access the Display object that corresponds to the screen (see Chapter 4) only from the point at which startApp() is invoked for the first time. Under a strict interpretation of the specification, therefore, initialization that involves a Display object cannot be performed in the constructor. Of course, actual MIDP implementations may not enforce this apparent restriction, but portability may be compromised if the MIDlet accesses the Display object in its constructor.

A MIDlet may refuse a request to be resumed from the Paused state by throwing a MIDletStateChangeException when its startApp() method is called, as described earlier.

When the host platform needs to terminate a MIDlet, it calls the MIDlet's destroyApp() method:

```
public abstract void destroyApp(boolean unconditional) throws
    MIDletStateChangeException;
```

In the destroyApp() method, the MIDlet should release all the resources that it has allocated, terminate any background threads, and stop any active timers. When the MIDlet is terminated this way, the unconditional argument has the value true, to indicate that the MIDlet cannot prevent the process from continuing. Under some circumstances, however, it is useful to give the MIDlet the option to not terminate, perhaps because it has data that it needs to save. In this case, the destroyApp() method can be invoked with the argument false, in which case the MIDlet can indicate that it wants to continue by throwing a MIDletStateChangeException. The following code illustrates how this technique can be used to implement the conditional shutdown of a MIDlet:

```
try {
    // Call destroyApp to release resources
    destroyApp(false);

    // Arrange for the MIDlet to be destroyed
    notifyDestroyed();
} catch (MIDletStateChangeException ex) {
    // MIDlet does not want to close
}
```

This code might be used to respond to an Exit button in the MIDlet's user interface. It begins by directly invoking the MIDlet's own destroyApp() method so that resources are released. If the MIDlet is not in an appropriate state to terminate, and destroyApp() is called with argument false, the MIDlet should throw a MIDletStateChangeException. The calling code should catch this exception and do nothing, as shown here. On the other hand, if the MIDlet is prepared to be terminate, it should complete the destroyApp() method normally, in which case the calling code uses the MIDlet notifyDestroyed() method to tell the MIDP platform that the MIDlet wants to be terminated.

FIFTH THIRD EX1059
Page 24 of 49

This example also illustrates the use of the `notifyDestroyed()` method, which is used by a MIDlet to voluntarily terminate. It is important to understand the relationship between the `destroyApp()` and `notifyDestroyed()` methods and when they are used:

- When the MIDlet is being destroyed by the platform, most likely because the user has requested it, the MIDlet's `destroyApp()` method is called with the argument `true`, and the MIDlet is destroyed when this method completes. It is not necessary in this case for the MIDlet to invoke its `notifyDestroyed()` method.
- When the MIDlet itself wants to terminate, typically because it has no more useful work to do or the user has pressed an Exit button, it can do so by invoking its `notifyDestroyed()` method, which tells the platform that it should be destroyed. In this case, the platform does not call the MIDlet's `destroyApp()` method; it assumes that the MIDlet is already prepared to be terminated. Most MIDlets invoke their own `destroyApp()` method to perform the usual tidy up before calling `notifyDestroyed()`, as shown earlier.

Note that calling `notifyDestroyed()` is the only way for a MIDlet to terminate voluntarily. MIDlets cannot terminate by calling the `System` or `Runtime` `exit()` methods, because these throw a `SecurityException`.

There are two other methods that a MIDlet may invoke to influence its own lifecycle:

```
public final void notifyPaused();
public final void resumeRequest();
```

The `notifyPaused()` method informs the platform that the MIDlet wishes to be moved to the Paused state; this has the same effect as if the platform had invoked the MIDlet's `pauseApp()` method. When the MIDlet calls `notifyPaused()`, the platform does *not* invoke its `pauseApp()` method, in the same way that it does not call `destroyApp()` in response to `notifyDestroyed()`, because it assumes that the MIDlet has prepared itself to be paused. A MIDlet often, therefore, precedes an invocation of `notifyPaused()` with a call to `pauseApp()` so that the appropriate steps are taken before the MIDlet is suspended.

The `resumeRequest()` method is the reverse of `notifyPaused()`; it tells the platform that a MIDlet in the Paused state wishes to return to the Active state. At some future time, the platform may resume the MIDlet by calling its `startApp()` method. The `resumeRequest()` method typically is called by a background thread or from a timer that the MIDlet left active while it was paused, an example of which is shown in the next section.

Developing MIDlets

To illustrate the MIDlet lifecycle and how it can be controlled, we'll create a very simple MIDlet that does the following:

- Prints a message when its constructor is called.
- Creates a timer that fires from time to time, putting the MIDlet in the paused state if it is active and returning it to the active state if it is paused. When the timer has been through this cycle twice, it terminates the MIDlet.

FIFTH THIRD EX1059
Page 25 of 49

- Creates a background thread when it is started that simply prints a message every second. This thread is allowed to run only when the MIDlet is active.

Since you haven't yet seen how to create user interfaces, this example MIDlet communicates by writing messages to its standard output stream. On a real device, you can't see what is written to standard output or standard error (unless you are using debug facilities provided by the device vendor), but most device emulators provide a way to monitor the content of these streams. There are several products available that allow you to build and test MIDlets either in an emulated environment or on the real device; some of these products are described in Chapter 9. Here, we'll use the Wireless Toolkit, which is available free of charge from Sun.

Building a MIDlet with the Wireless Toolkit

The Wireless Toolkit provides an implementation of MIDP together with an emulator that can be customized to look and behave somewhat like a number of real cell phones. It can also be used in conjunction with a third-party emulator that allows you to see how your MIDlets would behave on handhelds that are based on PalmOS. It is not, however, a complete development enviroment, because it does not provide an integrated editor to allow you to create, view, and modify source code. Consequently, if you want to use the Wireless Toolkit as part of a complete development cycle, you will need a text editor or IDE to manage the source code. At the time of writing, the Wireless Toolkit can be installed to integrate with Forte for Java, which is available for download from Sun's web site, and Borland JBuilder, but any IDE will do.

The first step when using the Wireless Toolkit is to create a project, which manages the source code, classes, and resources corresponding to a MIDlet suite. To do this, start the `KToolbar` and press the New Project button to open the New Project dialog, which is shown in Figure 3-5. For this example, the name of the MIDlet's main class should be `ora.ch3.ExampleMIDlet`, and the project name can be anything you like.

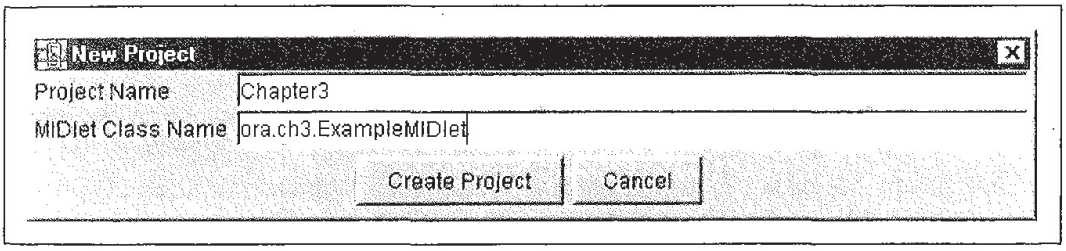


Figure 3-5: Creating a new project with the Wireless Tooklit

When you press the Create Project button in the dialog, the Wireless Toolkit opens another window, shown in Figure 3-6; it contains a set of tabs that allow you to provide the attributes used to generate the manifest for the MIDlet's JAR and the JAD file. You can edit these attributes by clicking the cell that you want to change and typing the new value. The fields on the Required tab contain the attributes shown in Table 3-2 that are marked as mandatory. Most of the values supplied by default can be used without modification. For example, the `MIDlet-Name` field (which is actually the name that will be used for the MIDlet suite, not for any individual MIDlet) matches the project name, and the name of the JAR that will be

FIFTH THIRD EX1059
Page 26 of 49

created is also derived from the project name. The only field you might want to change on this tab is `MIDlet-Vendor`, which is initially set to `Sun Microsystems` by default.

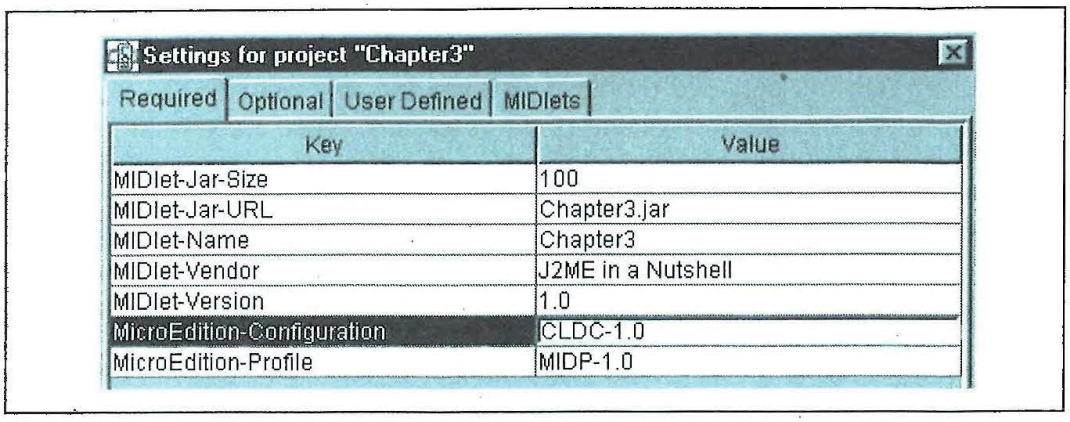


Figure 3-6: Setting required attributes for a MIDlet suite

To define the MIDlets that should be included in the MIDlet suite, select the MIDlets tab. Initially, this contains a single row whose content is constructed from the name of the project. In this example, the suite contains a single MIDlet called `ExampleMIDlet` in the package `ora.ch3`, so you should press the Edit button and edit the values for the `MIDlet-1` attribute on this tab so that it looks like this:

Key	*Name*	*Icon*	*Class*
`MIDlet-1`	`ExampleMIDlet`	`/ora/ch3/icon.png`	`ora.ch3.ExampleMIDlet`

In this example, the name assigned to the MIDlet matches the class name (ignoring the package prefix), but this need not be the case. Notice also that although the class name is specified in the usual way, with the parts of the name separated by periods, the location of the icon is specified as a filename, in which the path components are separated by a "/" character. If an icon is present, an absolute pathname must be provided here. If the MIDlet does not have an associated icon, this field should be left blank.

For a MIDlet suite with more than one MIDlet, you add an extra line for each MIDlet. It is important that consecutive numbers are used in the key field, so the next MIDlet to be added in this example would need to have the key `MIDlet-2`. Other required class files must be included in the JAR, but they should not be included in the MIDlets list.

For this example, we are also going to use a user-defined attribute. A user-defined attribute is a private attribute that can be set in the manifest and/or the JAD; its value can be retrieved at runtime by any MIDlet in the MIDlet suite. These attributes provide a mechanism similar to the setting of system properties in J2SE and allow the operation of the MIDlet to be customized without the need to recompile source code. In this example, we'll use a user-defined attribute to specify the length of a timer. To set the value of the attribute, select the User Defined tab and press the Add button. In the dialog box that appears, supply the property name as `Timer-Interval` and press OK. This creates a new entry in the table on the User Defined tab. Click in the Value cell, and type the required value,

FIFTH THIRD EX1059
Page 27 of 49

which, in this case, should be 3000. The property name is case-sensitive and, to avoid confusion with reserved attribute names, should not begin with "MIDlet-". The property value is always a string that is interpreted by the MIDlet. In this case, it represents the timer interval in milliseconds, so the value given here results in a timer that has a three-second interval. You'll see shortly how the MIDlet retrieves the values of user-defined attributes.

This completes the setting of the MIDlet's attributes. To save them, press the OK button at the bottom of the dialog. You can change these settings (perhaps to add extra MIDlets) at any time by pressing the Settings . . . button on the main KToolbar window, which is shown in Figure 3-7.

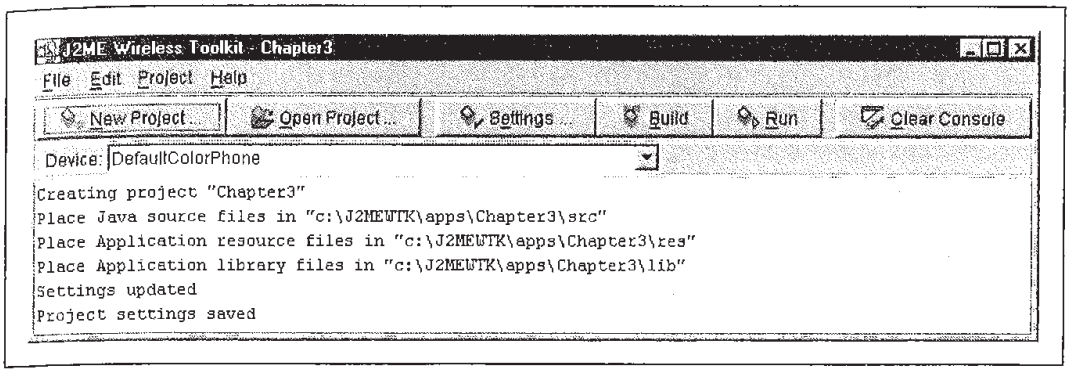


Figure 3-7: The main window of the Wireless Tooklit KToolbar

The next step is to place the source code and the icon for the MIDlet where the Wireless Toolkit can get access to them. Most IDEs allow you to choose where your project source files are kept, but the Wireless Toolkit uses a fixed filesystem layout for each project, based beneath the directory in which the Toolkit was originally installed. The name of the top-level directory for a project is derived from the name given to the project when it was created. If, for example, you installed the Windows Toolkit in the directory *c:\J2MEWTK*, all the files for the `Chapter3` project need to be placed below the directory *c:\J2MEWTK\apps\Chapter3*. When the `Chapter3` project was created, the toolkit created the following four directories below the main directory for the project:

src
: Holds the source code for the MIDlets and any shared classes

res
: Holds any resources required by the MIDlets, such as icons

lib
: Holds JAR or ZIP files for third-party libraries that the MIDlets need

bin
: Holds the JAR, JAD and manifest files

Before building the project, you need to place the appropriate files in the *src*, *res* and *lib* subdirectories. This example has one source file and a single icon, which can both be found in the directory *ora\ch3* of the source code for this book. The package structure used by the MIDlet must be reflected in the directory layout as seen by the Wireless Toolkit, as it would be by an IDE. Therefore, to install the files where the Wireless Toolkit can use them, you should copy them as follows,

FIFTH THIRD EX1059
Page 28 of 49

creating the *ora\ch3* subdirectory beneath both the *src* and *res* directories while doing so:

Source	*Destination*
ora\ch3\ExampleMIDlet.java	*c:\J2MEWTK\apps\Chapter3\src\ora\ch3\ExampleMIDet.java*
ora\ch3\icon.png	*c:\J2MEWTK\apps\Chapter3\res\ora\ch3\icon.png*

Once the files have been placed in the correct directories, the next step is to build the project by pressing the Build button on the KToolbar main window. The build process performs the following steps:

- Creates a *tmpclasses* directory below the main directory, compiles all the source files below the *src* directory, and places the class files beneath *tmpclasses*, having regard to the package hierarchy. Thus, for example, the class files for the MIDlet `ora.ch3.ExampleMIDlet` would be placed in the directory *c:\J2MEWTK\apps\Chapter3\tmpclasses\ora\ch3*.
- Creates a *classes* directory below the main directory and runs the preverifier on all of the classes found below *tmpclasses*, placing the verified class files below the *classes* directory in a directory layout that again reflects the package hierarchy. The verified *ExampleMIDlet* classes would, therefore, end up in *c:\J2MEWTK\apps\Chapter3\classes\ora\ch3*.
- Creates the manifest file and the JAD file and places them in the *bin* directory.



The source code for this book is actually stored in two different directory hierarchies, one for standard IDEs, the other for the J2ME Wireless Toolkit. This example showed you how to create a project from scratch using existing source files. A quicker way to use the book's source code is to copy the content of the directory *wtksrc* into *c:\J2MEWTK\apps*. This will give you subdirectories called *Chapter3*, *Chapter4*, etc., that contain all the source code and resources for each chapter's examples in the format expected by the J2ME Wireless Toolkit. To use each set of examples, select Open Project on the KToolBar main screen instead of Create Project, and then select the project from the dialog box that appears.

Running a MIDlet

At this stage, the JAR file has not been created, but you can nevertheless test the MIDlet suite by selecting an appropriate target device on the KToolbar main window and pressing the Run button. This loads the MIDlet classes, its resources, and any associated libraries from the *classes*, *res*, and *lib* subdirectories. If you select the default gray phone and press the Run button, the emulator starts and displays the list of MIDlets in this suite, as shown in Figure 3-8.

When the MIDlet suite is loaded, the device's application management software displays a list of the MIDlets that it contains and allows you to select the one you

FIFTH THIRD EX1059
Page 29 of 49



Figure 3-8: The Wireless Toolkit emulator

want to run. In this case, even though the suite contains only one MIDlet, the list is still displayed, as shown in Figure 3-8. Given the current lack of security for MIDlets imported from external sources, it would be dangerous for the device to run a MIDlet automatically, and, by giving the device user the chance to choose a MIDlet, it allows him the opportunity to decide not to run any of the MIDlets if, for any reason, they are thought to be a security risk or otherwise unsuitable. It is not obvious, though, on what basis such a decision would be made, since the user will see only the MIDlet names at this stage, but requiring the user to confirm that a MIDlet should be run transfers the ultimate responsibility to the user. In this case, the device displays the MIDlet name and its icon (the exclamation mark) as taken from the `MIDlet-1` attribute in the manifest file. The device is not obliged to display an icon, and it may use its own icon in preference to the one specified in the manifest.

When you run the MIDlet suite this way, the Wireless Toolkit compiles the source code with the option set to save debugging information in the class files, and it does not create a JAR file. If you want to create a JAR, you can do so by selecting the Package item from the Project menu. This rebuilds all the class files without debugging enabled, which reduces the size of the class files, a measure intended to keep the time required to download the JAR to a cell phone or PDA as small as possible. It also extracts the content of any JARs or ZIP files it finds in the *lib* subdirectory and includes them in the MIDlet JAR, after running the preverifier over any class files that it finds in these archives. The JAR can be used, along with the JAD file, to distribute the MIDlet suite for installation into a device over a network, as will be shown in the later section "Delivery and Installation of MIDlets."

Further information on the Wireless Toolkit and other MIDlet development environments can be found in Chapter 9.

FIFTH THIRD EX1059
Page 30 of 49

A Simple MIDlet

Now let's look at the implementation of the `ExampleMIDlet` class you have just built and packaged with the Wireless Toolkit. This simple MIDlet demonstrates the lifecycle methods that were described in the earlier section "MIDlet Execution Environment and Lifecycle," and it also illustrates how the MIDlet's foreground activity interacts with background threads, as well as how to create and use timers. The code for this example in shown in Example 3-1. For clarity, the timer-related code has not been included in the code listing; you'll see how that works when timers are discussed later in this chapter.

Example 3-1: A Simple MIDlet

```
package ora.ch3;

import java.util.Timer;
import java.util.TimerTask;
import javax.microedition.midlet.MIDlet;
import javax.microedition.midlet.MIDletStateChangeException;

public class ExampleMIDlet extends MIDlet {

    // Flag to indicate first call to startApp
    private boolean started = false;

    // Background thread
    private Thread thread;

    // Timer interval
    private int timerInterval;

    // Timer
    private Timer timer;

    // Task to run via the timer
    private TimerTask task;

    // Required public constructor. Can be omitted if nothing to do and no
    // other constructors are created.
    public ExampleMIDlet() {
        System.out.println("Constructor executed");

       // Get the timer interval from the manifest or JAD file.
        String interval = getAppProperty("Timer-Interval");
        timerInterval = Integer.parseInt(interval);
        System.out.println("Timer interval is " + interval);
    }

    protected void startApp() throws MIDletStateChangeException {
        if (!started) {
            // First invocation. Create and start a timer.
            started = true;
            System.out.println("startApp called for the first time");
            startTimer();
```


FIFTH THIRD EX1059
Page 31 of 49

Example 3-1: A Simple MIDlet (continued)

```
        } else {
            // Resumed after pausing.
            System.out.println("startApp called following pause");
        }

        // In all cases, start a background thread.
        synchronized (this) {
            if (thread == null) {
                thread = new Thread() {
                    public void run() {
                        System.out.println("Thread running");
                        while (thread == this) {
                            try {
                                Thread.sleep(1000);
                                System.out.println("Thread still active");
                            } catch (InterruptedException ex) {
                            }
                        }
                        System.out.println("Thread terminating");
                    }
                };
            }
        };
         thread.start()
    }

    protected void pauseApp() {
        // Called from the timer task to do whatever is necessary to pause
        // the MIDlet.
        // Tell the background thread to stop.
        System.out.println("pauseApp called.");
        synchronized (this) {
            if (thread != null) {
                thread = null;
            }
        }
    }

    protected void destroyApp(boolean unconditional)
                            throws MIDletStateChangeException {
        // Called to destroy the MIDlet.
        System.out.println("destroyApp called - unconditional = "
                            + unconditional);
        if (thread != null) {
            Thread bgThread = thread;
            thread = null;      // Signal thread to die
            try {
                bgThread.join();
            } catch (InterruptedException ex) {
            }
        }
        stopTimer();
```


FIFTH THIRD EX1059
Page 32 of 49

Example 3-1: A Simple MIDlet (continued)

```
    }

    // Timer code not shown here
}
```

This simple MIDlet does two things:

- Starts a background thread that writes a message to standard output every second so that you can see that the MIDlet is active
- Starts a timer that periodically pauses the MIDlet if it is active and makes it active again if it is paused

The code listing shows the implementation of the MIDlet's constructor and its `startApp()`, `pauseApp()` and `destroyApp()` methods. A MIDlet is not required to do anything in its constructor and may instead defer initialization until the `startApp()` method is executed. In this example, the constructor prints a message so that you can see when it is being executed. It also performs the more useful function of getting the interval for the timer that will be used to change the MIDlet's state. It is appropriate to put this code in the constructor because this value needs to be set only once. The timer value is obtained from the `Timer-Interval` attribute that was specified in the settings dialog of the Wireless Toolkit and subsequently written to the JAD file. Here is what the JAD file created for this MIDlet suite actually looks like:

```
MIDlet-1: ExampleMIDlet, /ora/ch3/icon.png, ora.ch3.ExampleMIDlet
MIDlet-Jar-Size: 100
MIDlet-Jar-URL: Chapter3.jar
MIDlet-Name: Chapter3
MIDlet-Vendor: J2ME in a Nutshell
MIDlet-Version: 1.0
Timer-Interval: 3000
```

A MIDlet can read the values of its attributes using the following method from the `MIDlet` class:

```
public final String getAppProperty(String name);
```

This method looks for an attribute with the given name; it looks first in the JAD file, and then, if it was not found there, in the manifest file. Attributes names are case-sensitive and scoped to the MIDlet suite, so every MIDlet in the suite has access to the same set of attributes. The `getAppProperty()` method can be used to retrieve any attributes in the JAD file or the manifest, so the following line of code returns the name of the MIDlet's suite, in this case `Chapter3`:

```
String suiteName = geAppProperty("MIDlet-Name");
```

The timer interval for this MIDlet is obtained as follows:

```
String interval = getAppProperty("Timer-Interval");
timerInterval = Integer.parseInt(interval);
```

Once the value in the form of a string has been retrieved, the next step is to convert it to an integer by calling the `Integer parseInt()` method. If the `Timer-Interval` attribute is not included in the JAD file or manifest (or if its name is misspelled), `getAppProperty()` returns `null`, and the `parseInt()`

FIFTH THIRD EX1059
Page 33 of 49

method throws an exception. A similar thing happens if the attribute value is not a valid integer. Notice that the constructor does not bother to catch either of these exceptions. The main reason for catching an exception is to display some meaningful information to the user and possibly allow recovery, but, strictly speaking, the MIDlet is not allowed to use the user interface in the constructor, so attempting to post a message would not necessarily work. The most appropriate thing to do in a real MIDlet is to install a default value for the timer interval and arrange to notify the user from the `startApp()` method, when access to the user interface is possible. In this simple example, we allow the exception to be thrown out of the constructor, which causes the MIDlet to be destroyed. Additionally, the version of MIDP in the Wireless Toolkit does, in fact, display the exception on the screen, but vendor implementations are not bound to do so.

Once the constructor has completed execution, the device eventually calls the MIDlet's `startApp()` method, which allocates any resources that the MIDlet needs. The `startApp()` method is also called when the MIDlet is resumed after being in the Paused state. In that case, however, it should allocate only the resources that were released by `pauseApp()`. A boolean variable called `started`, which is `false` only when `startApp()` is entered for the first time, is used to distinguish these two cases:

- When `started` is `false`, `startApp()` creates and starts the MIDlet timer and the MIDlet's background thread.
- When `started` is `true`, `startApp()` does not need to concern itself with the timer, because it is not canceled by the `pauseApp()` method. It does, however, create a new background thread, because the original thread will be stopped when the MIDlet is paused.

Since the timer is going to be active throughout the lifetime of the MIDlet, it could have been allocated in the constructor. We deferred creating the timer until `startApp()` executes for the first time, however, because it isn't actually needed until that point; it is better, in an environment with such limited memory, to delay allocating resources until they are needed. The decision whether to commit resources in the constructor or in the `startApp()` method depends on the MIDlet and must therefore be made on a case-by-case basis.

The `pauseApp()` method is relatively simple. Its job is to release any resources that the MIDlet does not need while it is not in the Active state. The MIDlet is making use of only two resources:

- A background thread printing a message every second
- A timer responsible for pausing and resuming the MIDlet periodically

Clearly, we can't stop the timer when the MIDlet is paused, because the timer is responsible for resuming it later. Therefore, the only resource the `pauseApp()` method can release is the background thread, by arranging for it to stop execution.

How is the `pauseApp()` method going to stop the background thread? The J2SE `Thread` class has two methods that might help: `stop()` and `interrupt()`. Neither of these methods is available in the CLDC version of `Thread`, however, so it is not possible to act directly on the background thread to stop it. Instead, we

FIFTH THIRD EX1059
Page 34 of 49

use a common mechanism, a shared variable that the thread inspects from time to time to find out whether it has been asked to stop. In this case, the MIDlet class keeps a reference to the `Thread` instance in a variable called `thread`. In order to stop the thread, the `pauseApp()` method sets this variable to `null`, while the main loop of the background thread checks its value on each pass:

```
public void run() {
    System.out.println("Thread running");
    while (thread == this) {
        try {
            Thread.sleep(1000);
            System.out.println("Thread still active");
        } catch (InterruptedException ex) {
        }
    }
    System.out.println("Thread terminating");
}
```

You'll notice that this code actually checks not whether the `thread` variable is `null`, but whether it is pointing to the background thread itself. This prevents a race condition in which the `pauseApp()` method clears `thread` to `null`, and the timer thread resumes the MIDlet before the background thread restarts following the `sleep()` call and checks its value. In this case, the `startApp()` method has started a new thread and stored its reference in `thread`, which therefore will not be `null` when the previous code checks it.

Finally, the `destroyApp()` method needs to stop the background thread and stop and release the timer. The thread can be stopped just as it is in the `pauseApp()` method. However, `destroyApp()` also waits for the thread to terminate so that it can guarantee that the MIDlet is not using any resources when it returns. It does this by calling the `Thread.join()` method, which blocks until the thread terminates (and returns immediately if it has already terminated). The `stopTimer()` method, which `destroyApp()` calls to stop and release the timer, is described in the next section.

If you now launch the MIDlet from the emulator, you'll see the results in the Wireless Toolkit's console window, an extract of which follows:

```
Constructor executed
Timer interval is 3000
startApp called for the first time
Timer started.
Thread running
Thread still active
Thread still active
Timer scheduled
>> Pausing MIDlet
pauseApp called.
Thread still active
Thread terminating
Timer scheduled
>> Resuming MIDlet
startApp called following pause
Thread running
```


FIFTH THIRD EX1059
Page 35 of 49

As you can see, the constructor is executed first; it reads the value of the timer interval from the JAD file. Then `startApp()` is called, and it detects that it is being called for the first time and starts both the timer and the background thread. The "Thread running" and "Thread active" messages are printed by the background thread itself and show that the thread executes its loop twice before the timer fires. The code that executes when the timer expires, which will be shown in the next section, alternately pauses and resumes the MIDlet. In this case, as you can see, `pauseApp()` is called, which signals the background thread to stop running; the "Thread terminating" message indicates that the thread detects that it has been told to stop. Three seconds later, the timer expires again and resumes the MIDlet, causing its `startApp()` method to be invoked again to recreate the background thread. This process continues through two cycles, at which point the timer code destroys the MIDlet.

Timers and TimerTasks

Code to be executed when a timer expires should be implemented as a `TimerTask` and scheduled by a `Timer`. The `Timer` class provides the ability to execute sequentially one or more `TimerTasks` in a dedicated background thread. Usually, a MIDlet creates a single `Timer` to schedule all its `TimerTasks`, but it is possible to have more than one `Timer` active, each running its assigned `TimerTasks` in its own thread.

`TimerTask` is an abstract class with three methods:

```
public abstract void run();
public boolean cancel();
public long scheduledExecutionTime();
```

You create a unit of work to be scheduled by a `Timer` by subclassing `TimerTask` and implementing the `run()` method. You can schedule the `run()` method to be executed just once or to be executed repeatedly at either a fixed interval or a fixed rate. You can use the `TimerTask cancel()` method to stop future execution of a specific `TimerTask`. You may invoke it from the `run()` method, in which case the current execution of the task is allowed to complete, or you make invoke it from somewhere else. This method returns `true` if the task was scheduled to run either once or repeatedly and has been canceled; it returns `false` if the task was not associated with a `Timer` or if it had had been scheduled to be run once and has already run. The `scheduledExecutionTime()` method gets the time at which the task was most recently executed by its associated `Timer`. If called from within the `run()` method, it returns the time at which the `run()` method began execution. The value returned by this method is the number of milliseconds since midnight, January 1, 1970, which is the same as that returned by the `System currentTimeMillis()` method. If this method is called before the task is scheduled for the first time, its return value is undefined.

The `Timer` class has two methods that can be used to arrange for a task to be run exactly once:

```
public void schedule(TimerTask task, Date time);
public void schedule(TimerTask task, long delay);
```


FIFTH THIRD EX1059
Page 36 of 49

The first of these methods schedules the task at the given time or as soon as possible afterwards; the second runs the task when a given time interval, specified in milliseconds, has passed. There are four methods that schedule a task for repeated execution:

```
public void schedule(TimerTask task, Date time, long period);
public void schedule(TimerTask task, long delay, long period);
public void scheduleAtFixedRate(TimerTask task, Date time, long period);
public void scheduleAtFixedRate(TimerTask task, long delay,
    long period);
```

The difference between these methods is that the first two apply a fixed delay between successive executions of the task, and the last two attempt to execute the task at a fixed rate. In both cases, the desired interval between task executions is given by the `period` parameter. Figure 3-9 shows how fixed-delay and fixed-rate scheduling differ.

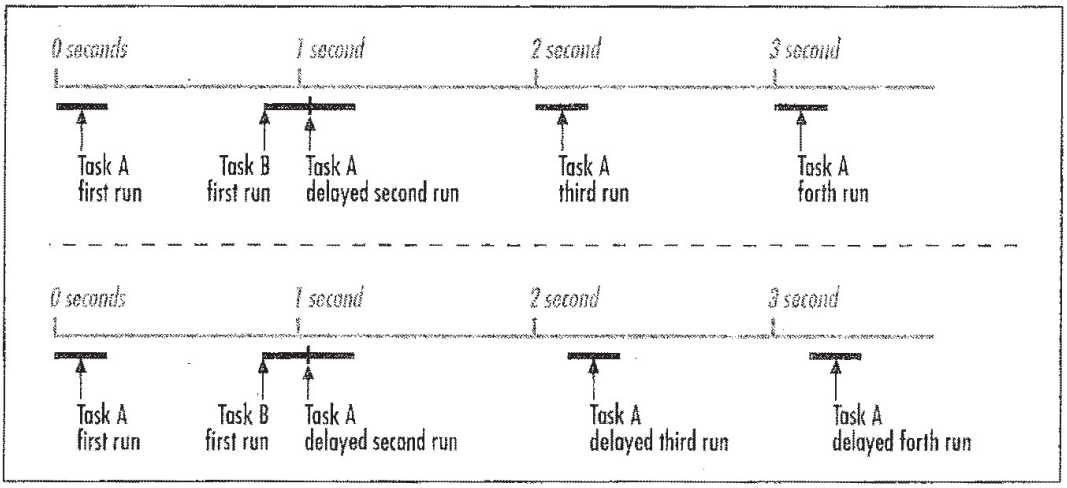


Figure 3-9: Fixed-delay (top) and fixed-rate scheduling of TimerTasks

In this example, task A is scheduled to run once every second; task B runs once, starting 900 milliseconds along the time line shown in the diagram. Task A first runs at T+0, followed by task B, which begins its execution at T+900ms. Task B takes 200 milliseconds to complete, however, which means that it is still running at T+1 second, when task A is supposed to run for the second time. Since a `Timer` can schedule only one `TimerTask` at a time, the execution of task A is delayed until task B finishes. Task A's second run begins, therefore, at T+1100ms. The difference between fixed-delay and fixed-rate scheduling is what happens as a result of this delay:

- In fixed-rate scheduling, the next execution of task A is scheduled relative to the time its previous execution should have started. In this case, task A should have begun execution at T+1 second. Under fixed rate scheduling, it will next run at T+2 seconds, as it would have had task B not delayed it.
- With fixed-delay scheduling, the next execution of task A is timed relative to the time its previous execution actually took place. Since task A last ran at T+1100ms, it will next run at T+2100ms, then at T+3100ms, and so on.

FIFTH THIRD EX1059
Page 37 of 49

With fixed-delay scheduling, therefore, any delay affects all future executions of the task. With fixed-rate scheduling, however, an attempt is made to "ignore" the delay and schedule the task again where it would have run had there been no delay.

In some cases, additional executions of a fixed-rate task may be required to ensure that it runs the correct number of times when viewed over a long period. When this is necessary, the task may be run two or more times in succession to catch up with the number of times that it should have been run. For example, fixed-rate scheduling would be appropriate if you were using a timer to trigger redrawing the second hand of a clock displayed on the screen. Delayed execution of the redrawing task would cause the second hand to move more slowly, but the extra executions would ensure that it eventually moved forward to catch up with the real time. By contrast, using fixed delay execution in this case would result in the clock losing time that it would never make up, because execution delays are never corrected.

You may be able to reduce timing delays by using more than one `Timer` and dividing tasks among the `Timers`, because each `Timer` uses its own `Thread`. This only works, however, if the platform has more than one processor (which is unlikely in a J2ME environment), or if it has preemptive thread scheduling and chooses to suspend the thread of the `Timer` scheduling the long-running task B in favor of the thread for task A's `Timer`. The most reliable way to obtain predictable timer scheduling, however, is to ensure that code to be executed by a `TimerTask` executes as quickly as possible and does not block.

Like `TimerTask`, the `Timer` class has a `cancel()` method:

```
public void cancel();
```

This method cancels all the `TimerTasks` associated with the `Timer`. The `Timer`'s thread stops executing when it has no more `TimerTasks` to be scheduled and there are no live references to it.

Example 3-2 shows the timer-related code for our example MIDlet.

Example 3-2: Using a MIDlet Timer

```
// Starts a timer to run a simple task
private void startTimer() {

    // Create a task to be run
    task = new TimerTask() {
        private boolean isPaused;
        private int count;

        public void run() {
            // Pause or resume the MIDlet.
            System.out.println("Timer scheduled");
            if (count++ == 4) {
                // Terminate the MIDlet
                try {
                    ExampleMIDlet.this.destroyApp(true);
                } catch (MIDletStateChangeException ex) {
```


FIFTH THIRD EX1059
Page 38 of 49

Example 3-2: Using a MIDlet Timer (continued)

```
                    // Ignore pleas for mercy!
                }
                ExampleMIDlet.this.notifyDestroyed();
                return;
            }
            if (isPaused) {
                System.out.println(">> Resuming MIDlet");
                ExampleMIDlet.this.resumeRequest();
                isPaused = false;
            } else {
                System.out.println(">> Pausing MIDlet");
                isPaused = true;
                ExampleMIDlet.this.pauseApp();
                ExampleMIDlet.this.notifyPaused();
            }
        }
    };

    // Create a timer and schedule it to run
    timer = new Timer();
    timer.schedule(task, timerInterval, timerInterval);
    System.out.println("Timer started.");
}

// Stops the timer
private void stopTimer() {
    if (timer != null) {
        System.out.println("Stopping the timer");
        timer.cancel();
    }
}
```

The `startTimer()` method, which is called during the first invocation of `startApp()`, creates a `TimerTask` and schedules it to be run by a `Timer` object with the initial delay and repeat period given by the `Timer-Interval` attribute obtained from the application descriptor. The `stopTimer()` method is called from `destroyApp()`. It cancels the `TimerTask` and the `Timer` by calling the `Timer`'s `cancel()` method.

The code that is executed when the timer expires is worth looking at because it demonstrates how to control the lifecycle of a MIDlet. The intent of this code is to pause the MIDlet if it is active when the timer expires and resume if it is paused. However, there is no method that allows a MIDlet to find out whether it is in the Paused state, so the timer code has to retain this state for itself using an instance variable called `isPaused`. The code used to suspend the MIDlet looks like this:

```
isPaused = true;
ExampleMIDlet.this.pauseApp();
ExampleMIDlet.this.notifyPaused();
```

The `notifyPaused()` method tells the MIDlet scheduler that the MIDlet wants to be moved into the Paused state. As stated earlier, when the MIDlet calls this method, it is assumed that it is ready to be suspended, so its `pauseApp()` method

FIFTH THIRD EX1059
Page 39 of 49

is not called to give it a chance to release resources. For this reason, the timer code calls the MIDlet's `pauseApp()` method directly before suspending it. Moving a MIDlet to the Paused state simply means that it no longer has access to the screen and so does not receive user interface events in response to key presses or pointer movements. Timers and background threads belonging to a suspended MIDlet continue to be scheduled, provided that they are not stopped by the MIDlet itself in its `pauseApp()` method.

Moving the MIDlet from the Paused state to the Active state is a little easier:

```
ExampleMIDlet.this.resumeRequest();
isPaused = false;
```

The `resumeRequest()` call notifies the scheduler that the MIDlet would like to be made Active. In response to this, the MIDlet's `startApp()` method will be called at some future time to allow it to reallocate resources that were released when it was paused. If another MIDlet is currently in the foreground, the resumed MIDlet has to wait until the foreground MIDlet is paused or terminates before it becomes eligible to become the foreground MIDlet and recover use of the screen and input devices.

Finally, after two suspend/resume cycles are completed, the timer code destroys the MIDlet by calling `notifyDestroyed()`:

```
// Terminate the MIDlet
 try {
    ExampleMIDlet.this.destroyApp(true);
} catch (MIDletStateChangeException ex) {
    // Ignore pleas for mercy!
}
ExampleMIDlet.this.notifyDestroyed();
```

As is the case with `notifyPaused()`, the MIDlet's `destroyApp()` method is not invoked as a result of a call to `notifyDestroyed()`, so the timer code explicitly invokes it in order to allow the MIDlet to release its resources. Because this is an involuntary termination, the `destroyApp()` method is called with its `unconditional` argument set to `true`. However, care is taken to catch a `MIDletStateChangeException` in case the `destroyApp()` method ignores this argument. It is important to note that `notifyDestroyed()` does not actually terminate the MIDlet or any of its threads; it simply arranges for the MIDlet never to be scheduled as the foreground MIDlet and removes it from the list of active MIDlets. It is the MIDlet's responsibility to stop its active threads and timers in its `destroyApp()` method. Failure to do this may cause the Java VM to continue running and consuming memory when it has no useful work to do, which is unacceptable given the resource constraints of the typical MIDP device.

Delivery and Installation of MIDlets

The MIDP specification creates the concept of a MIDlet, defines its lifecycle and its execution environment, and specifies the programming interfaces that a MIDlet can expect to be present on any conforming device. However, it currently does not address in any detail how the user should locate MIDlet suites, how MIDlet suites will be installed on a cell phone or a PDA, and what facilities are to be provided to allow the user to select and launch an installed MIDlet or to remove

FIFTH THIRD EX1059
Page 40 of 49

MIDlet suites from the device. These features are not covered in detail in the MIDP specification because they are largely device-specific. Instead, it refers loosely to software that is intended for application delivery and management. The term *Application Management Software* (AMS) is generally used to describe the software components that take on this responsibility.* The MIDP reference implementation provides an example AMS for the benefit of vendors porting the software to their own devices, and both the Wireless Toolkit and the MIDP for PalmOS product have their own AMS implementations, which allow software to be installed from two different sources:

From a local host computer via a dedicated, relatively high speed connection
: This mode of operation is particularly suitable for PDAs, which are typically associated with a desktop or laptop computer with which they periodically synchronize. Synchronizing backs up the user's data from the handheld onto the larger system and copies software and data in the other direction, as well. The MIDP for PalmOS implementation is a good example of this, because its AMS allows MIDlet suites to be installed from a host PC during the synchronization process. Once the MIDlets are installed, they can be launched on the PDA in the same way as its native applications. The same application management features are supported for MIDlets, so they appear to be almost the same as native applications.

Over a network to which the device is connected
: This is the most common way in which MIDlets are downloaded to cell phones and similar wireless devices, although it is also applicable to network-connected PDAs. The process of deploying MIDlet suites over a network is referred to as *over-the-air provisioning*, or *OTA provisioning* for short. OTA provisioning is not part of the MIDP specification, but it is likely to be the dominant mechanism for distributing MIDlets, and it will doubtless be included in the formal specification in the near future. An AMS that supports installation of MIDlets from an HTTP server is included in the Wireless Toolkit.

Over-the-Air Provisioning

With OTA provisioning, MIDlet providers install their MIDlet suites on web servers and provide hypertext links to them. A user activates the links to download the MIDlets to a cell phone via a WAP or Internet microbrowser. Figure 3-10 shows the steps involved in a typical MIDlet installation.



OTA provisioning as described in this section is not formally a part of the MIDP specification at the time of writing, but it is likely to be included in the next version of the specification. Meanwhile, it has the status of best-practice recommendation.

* The term *Java Application Manager* (or JAM) was originally used to describe the MIDP application management software. However, this resulted in confusion with the Java Application Manager software that is part of the CLDC reference implementation, which performs similar functions but with which it is incompatible.

FIFTH THIRD EX1059
Page 41 of 49

The process begins when the user fetches a page from the corporate web site of the (fictional) corporation ACME, Inc. The page includes a link to a suite of MIDlets that allow the user to browse ACME's product catalog and place orders directly from a Java-enabled cell phone. Intrigued by this prospect, the user activates the link, which causes a request for the target to be sent to ACME's web server. The link in question would look something like this:

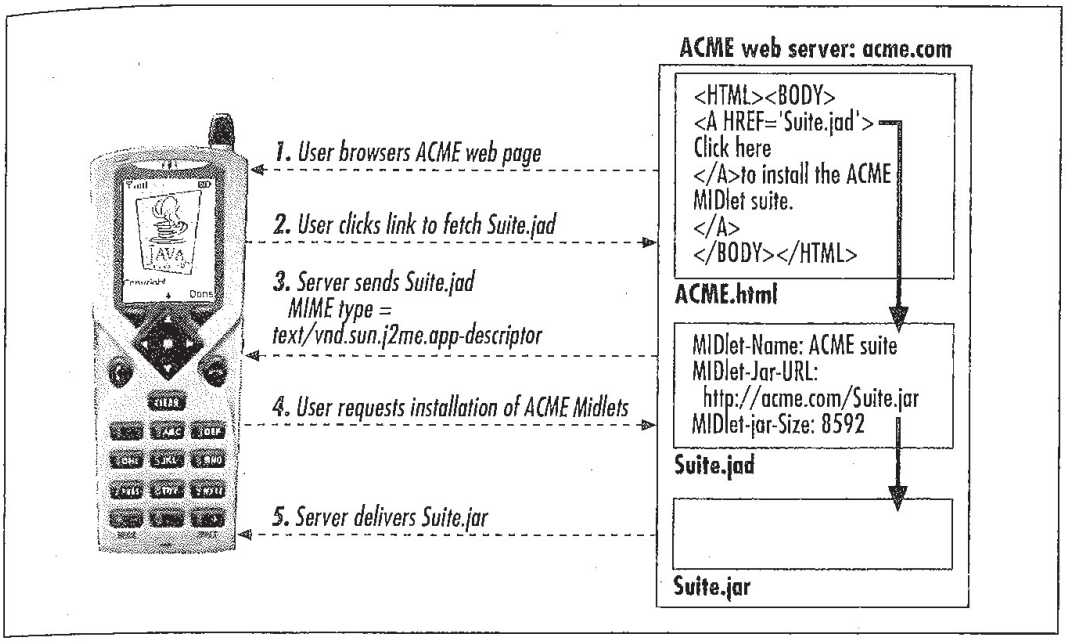


Figure 3-10: Installing MIDlets using OTA provisioning

```
<A HREF="Suite.jad">Click here</A> to install the ACME MIDlet suite
```

As you can see, this link points to the JAD file for the ACME MIDlet suite. The request to retrieve this file is sent by the cell phone's browser (see step 2 in Figure 3-10), but it will be passed to and handled by the phone's application management software. To enable browsers to easily identify JAD files, the web server is configured to return them with the following MIME type:

```
text/vnd.sun.j2me.app-descriptor
```

On receipt of data with this content type, the phone's AMS activates and displays the content of the application descriptor, so that the user can decide whether or not to install the MIDlet suite. At this stage, the user has waited only a relatively short time for the download of the small JAD file. Since this file contains an attribute that corresponds to the size of the JAR file that contains the MIDlets as well as a textual description of the services they provide, the user should be able to choose whether to install them. This is the advantage of providing MIDlet information in both the JAD file and the JAR file manifest.

Should the user decide to install the MIDlets, the AMS looks for the `MIDlet-Jar-URL` attribute in the JAD file and sends a request to that URL for the JAR, which the server should return tagged with the MIME type `application/java-archive`.

At this point, the MIDlet suite is installed, and the user can select and run the individual MIDlets. Following installation, the AMS may be required to deliver a status

FIFTH THIRD EX1059
Page 42 of 49

report to the provisioning server indicating whether the suite was successfully installed and identifying the reason for failure if it was not. This report takes the form of a status code and a status message that is sent using an HTTP `POST` request to the URL given by the `MIDlet-Install-Notify` attribute in the JAD file. If this attribute is not present, no installation report is sent. Of course, the server must be configured to expect an installation report at the given URL. The server typically uses a servlet or CGI script to save the report along with details of the originator for later use.



If you are not familiar with the HTTP protocol, you'll find a discussion of those parts of it that are supported by MIDP devices, including the POST request, in Chapter 6. More complete coverage of HTTP can be found in *Java Network Programming* by Elliotte Rusty Harold (O'Reilly & Associates, Inc.).

The status codes and their meanings are listed in Table 3-3.

Table 3-3: Status Codes Used to Report Success or Failure of MIDlet Installation

Status Code	*Meaning*
900	Success
901	Insufficient memory
902	Canceled by the user
903	Loss of network service (because of the network service loss, this report may never get delivered to the server)
904	JAR size mismatch
905	Attribute mismatch
906	Invalid descriptor

As well as implementing the MIDlet discovery and installation service as just described, the AMS software is required to provide the following functionality:

MIDlet suite updates

MIDlet updates are delivered just as the original MIDlet suite is: the user returns to the original server and requests the software as if an installation were being performed. Because the JAD file contains the version number of the associated MIDlet suite, the AMS can determine whether the software already installed is older than that on the server; if it is, the AMS can perform an upgrade, with permission from the user. Equally important, it can avoid downloading the JAR file if the newest version is already installed.

MIDlet selection and execution

The AMS provides the user with a means of selecting an installed MIDlet to run. The exact means by which this is achieved is device-dependent. On a cell phone, a menu item might give the user the ability to launch the AMS, or individual MIDlet suites may be included in the menu itself. On a PDA, MIDlet suites might be available in exactly the same way as native applications.

FIFTH THIRD EX1059
Page 43 of 49

MIDlet removal

The Java application management software is responsible for removing MIDlet suites from the device on user request. MIDlets cannot be removed individually. Following successful removal, the application manager must also delete any persistent storage resources that were allocated to the MIDlet suite (see "Persistent Storage" in Chapter 6 for further details). Because MIDlet removal causes loss of persistent data and is therefore almost certainly an irreversible process, the AMS will normally prompt the user for confirmation. The MIDlet suite vendor can use the `MIDlet-Delete-Confirm` attribute in the JAD file to include a message that should be displayed to the user before removal. This message can be used to warn the user of the consequences, if any, of removing the MIDlet suite.

The Wireless Toolkit Application Management Software

To prepare a MIDlet suite for remote installation, take the following steps:

1. Install the MIDlet suite JAR file on your web server.
2. Edit the JAD file so that its `MIDlet-Jar-URL` attribute points to the JAR file. Note that the specification requires that an *absolute* URL is required in the JAD file; relative URLs are not guaranteed to work. The Wireless Toolkit does not generate a JAD file containing an absolute URL, so you will need to edit it manually.
3. Place the JAD file on the web server.
4. Create an HTML or WML page with a hypertext link to the JAD file. The hypertext link must use an absolute URL, since application managers are not required to support relative URLs.
5. Configure the web server so that JAD files are returned with MIME type `text/vnd.sun.j2me.app-descriptor` and JAR files with MIME type `application/java-archive`.

The Wireless Toolkit contains a graphical AMS that can be used to test the OTA provisioning of MIDlet suites as well as to provide developers and vendors with a demonstration of typical application management and removal features. To use it, run the emulator provided with the Toolkit from the command line and pass it the argument *–Xjam*. Assuming you have installed the Wireless Toolkit in the directory *c:\j2mewtk*, issuing the following command in DOS starts the emulator and activates the AMS:

```
c:\j2mewtk\bin\emulator.exe -Xjam
```

When started, the application manager displays the Java logo and a copyright message. Press the Done button to show the application manager's main screen, which is shown on the left of Figure 3-11.

Pressing the Install button opens another screen that allows you to supply the URL of an HTML page that contains links to MIDlet suites, as shown on the right of Figure 3-11. This should be the URL of the HTML page set up previously, in step 4. The directory *ora\ch3* in this book's example source code contains a sample

FIFTH THIRD EX1059
Page 44 of 49

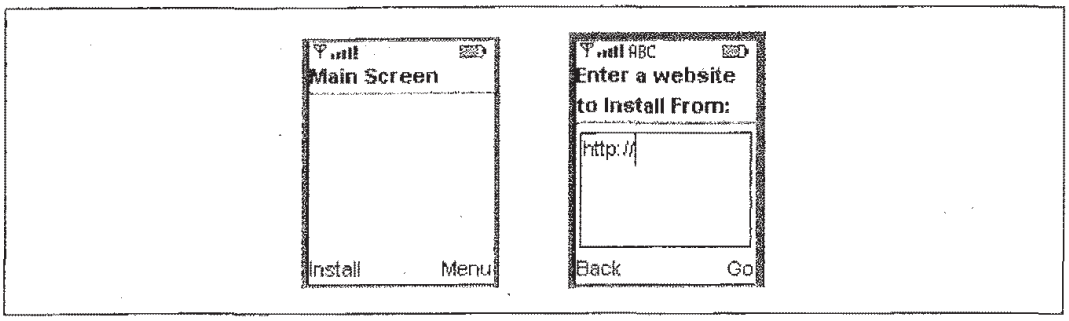


Figure 3-11: The Wireless Toolkit application management software

HTML file called *MIDlet.html* that you can use for testing purposes. You should compile and package the MIDlet in this directory in the usual way and copy the files *MIDlet.html*, *Chapter3.jad* and *Chapter3.jar* onto your web server. Open *Chapter3.jad* and change the `MIDlet-Jar-URL` attribute to the absolute URL that corresponds to the location of the JAR file. Also edit the *MIDlet.html* file so that the `HREF` attribute in the `<A>` tag is the absolute URL of the JAD file.

Press the Go button to start the process. At this point, the AMS loads the HTML page and scans it for links that point to JAD files. A commercial application manager distinguishes these links from other links by making a request to the server for the target of the link and looking for a returned MIME type of `text/vnd.sun.j2me.app-descriptor`. However, the Wireless Toolkit AMS appears to take a shortcut and simply looks for links for which the target URL ends with *.jad*. If the target page does not contain any links that correspond to MIDlet suites, the error message shown on the left of Figure 3-12 appears.

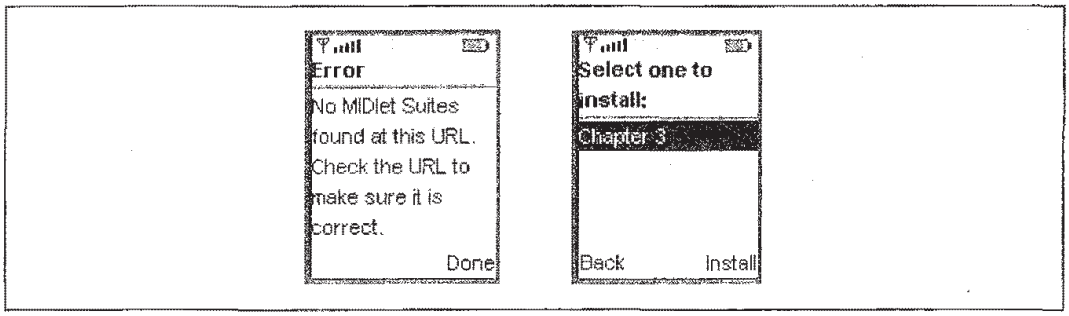


Figure 3-12: Selecting a MIDlet suite for installation

If you experience problems with this example, check that the server is properly configured and that the JAR and JAD files are consistent:

- The URL that you supply to the AMS must point to an HTML file that contains *absolute* hypertext links to one or more JAD files. The HTML file for this example looks like this:

```
<HTML>
    <HEAD>
        <TITLE>
        J2ME in a Nutshell Example MIDlet Download Page
        </TITLE>
    </HEAD>
    <BODY>
```


FIFTH THIRD EX1059
Page 45 of 49

```
        Install the example MIDlet suite for
        <A HREF="http://localhost:8080/Chapter3.jad">Chapter 3</A> of
"J2ME in a Nutshell".
    </BODY>
</HTML>
```

- The web server must be configured to return JAD files with MIME type `text/vnd.sun.j2me.app-descriptor`.
- The `MIDlet-Jar-URL` attribute in the JAD file must be an absolute URL pointing to the JAR file.
- The JAD file must contain the mandatory attributes listed in Table 3-2.

If the AMS locates any JAD files, it displays a list of the links that point to them, using the text within the `<A></A>` tag pair to identify each MIDlet suite, as shown on the right side of Figure 3-12. This implementation does not display the MIDlet suite name or the JAR file size from the JAD file because it hasn't fetched it yet. To continue with the installation process, press the Install button.

At this point, the Wireless Toolkit AMS reads the JAD file from the server and uses the `MIDlet-Jar-URL` attribute to locate and fetch the JAR file. If this process succeeds, a confirmation message appears as shown on the left side of Figure 3-13. After a short pause, the application manager switches back to its main screen, shown in the center of Figure 3-13. If you compare this to Figure 3-11, you'll see that the main screen now contains the name of the MIDlet suite that was just installed. The list of installed MIDlets is saved on the device, so this list will reappear when you next run the emulator. In the case of a MIDlet suite containing more than one MIDlet, the list in the main screen displays each suite together with the MIDlets that it contains. An example of this is shown on the right-hand side of Figure 3-13, in which a MIDlet suite called `Chapter5` containing individual MIDlets called `Socket`, `Time`, `Ranking` and others that are not visible in the screen shot, has been installed.

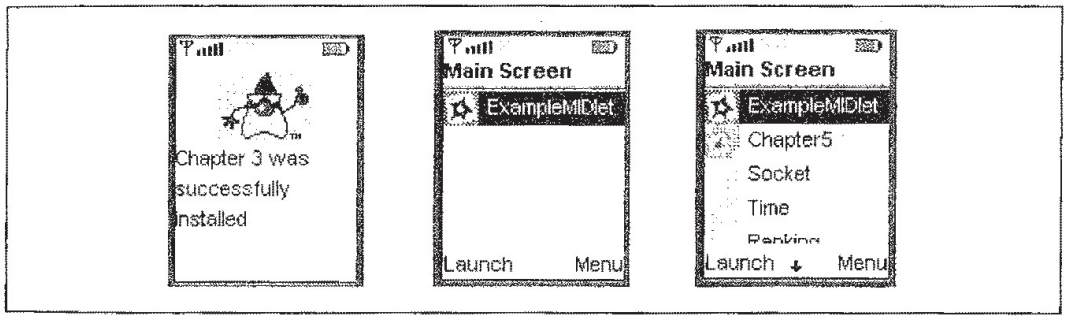


Figure 3-13: Installing a MIDlet suite

The main screen also includes an option that lets you launch MIDlets. If you select a MIDlet suite and choose this option, the usual MIDlet selector lets you pick the actual MIDlet to be run (see Figure 3-8). For a suite with multiple MIDlets, you can also choose an individual MIDlet from the main screen and launch it directly.

The Menu option provides access to the other application management features of the Wireless Toolkit AMS, presented in the form of a list, as shown in Figure 3-14.

FIFTH THIRD EX1059
Page 46 of 49

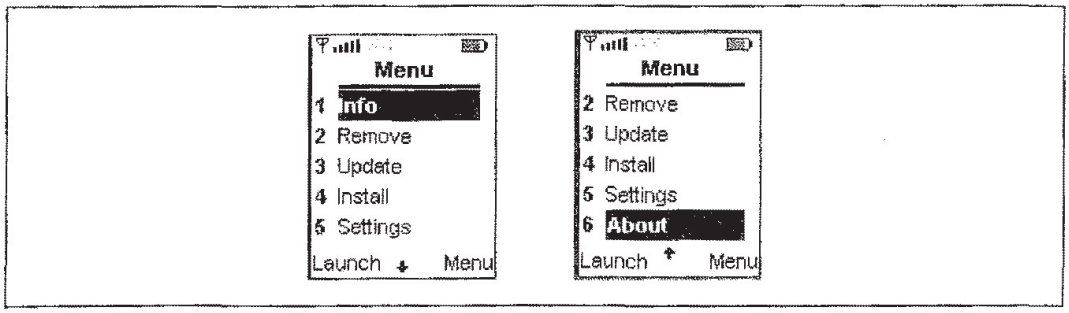


Figure 3-14: The Wireless Toolkit AMS application management menu

Of these menu items, only the first three are worth discussing here. Each of these items operates on a MIDlet suite, so selecting any of them brings up another copy of the MIDlet selection screen so that you can choose the suite to which the command should apply.

The Info command displays the content of the JAD file that was fetched when the MIDlet suite was installed. Ideally, this information would be displayed to the user before the installation process starts, but, as noted previously, the Wireless Toolkit AMS does not implement this feature. Figure 3-15 shows the information displayed for the `Chapter3` MIDlet suite.

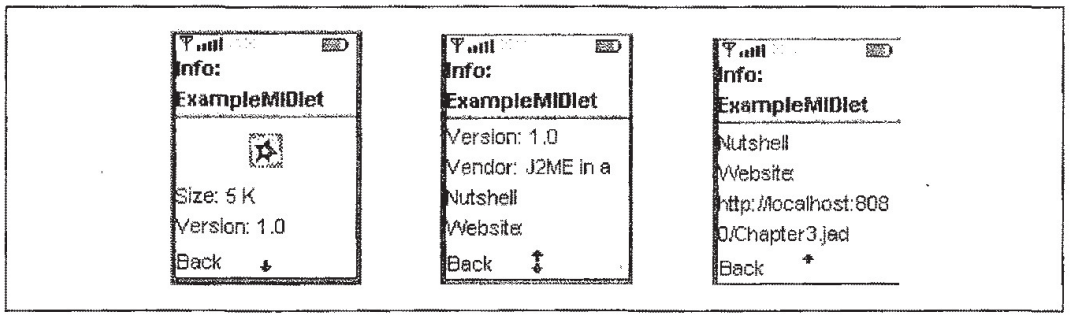


Figure 3-15: MIDlet suite information as displayed by the Windows Tooklit AMS

The Update command reinstalls the MIDlet suite from its original source. As noted earlier, the AMS can compare the MIDlet version in its installed JAD file and the one it acquires from the server to determine if it already has the latest version of a MIDlet suite.

The Remove option deletes a MIDlet suite and all its associated persistent storage from the device. The Wireless Toolkit AMS displays a warning message and asks the user for confirmation before performing this operation (see Figure 3-16).

The Wireless Toolkit AMS can be controlled from the command line as well as through its user interface. For example, you can install a MIDlet suite directly from a web server using the command:

```
c:\j2mewtk\bin\emulator.exe -Xjam:install=http://www.yourserver.com/
    SOMETHING/Chapter3.jad
```

A complete description of the command-line arguments recognized by the Wireless Toolkit emulator can be found in Chapter 8.

FIFTH THIRD EX1059
Page 47 of 49

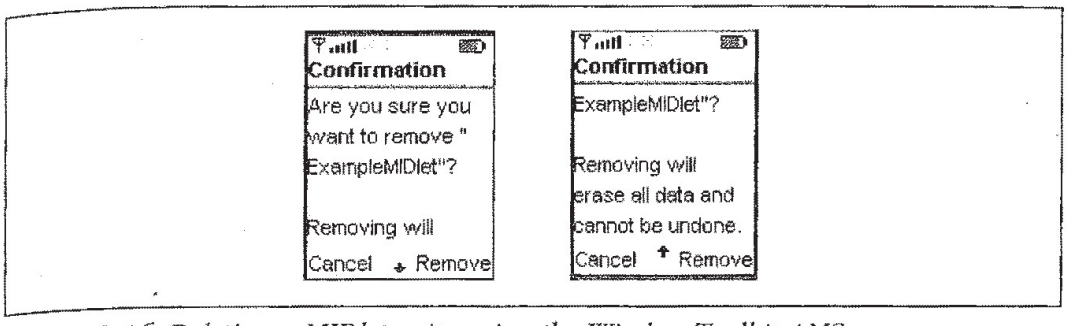


Figure 3-16: Deleting a MIDlet suite using the Wireless Toolkit AMS

FIFTH THIRD EX1059
Page 48 of 49

Java

J2ME IN A NUTSHELL



J2ME in a Nutshell is an indispensable quick reference for programmers working with the Java 2 Micro Edition (J2ME). J2ME is a new family of specifications from Sun Microsystems, Inc., that defines compact versions of the standard Java 2 platform. J2ME can be used to develop Java applications for resource-constrained devices such as cell phones, PDAs, two-way pagers, and set-top boxes.

Specifically, this book:

- Introduces the J2ME platform and development environments such as the Java Wireless Toolkit.
- Provides detailed coverage of the capabilities and requirements of the CLDC, MIDP, and MIDlets.
- Offers an in-depth look at MIDlet user interfaces, as well as tips and tricks for to working with the MIDP UI APIs.
- Explains how to use the Generic Connection Framework APIs for wireless networking across the Internet, as well as APIs for persisting data with the MIDP Record Management System (RMS).
- Presents an introduction to the CDC and its profiles.

This book also includes O'Reilly's trusted classic-style quick-reference material for all the classes in the various J2ME packages, including *java.lang*, *java.io*, *java.util*, *javax.microedition.io*, *javax.microedition.lcdui*, *javax.microedition.midlet*, and *javax.microedition.rms*. Once you've started working with J2ME, you'll keep this book next to your keyboard as a handy reference while you program.

This book is part of a multi-volume set of quick references that every Java programmer needs. It is a companion to both *Java in a Nutshell* and *Java Foundation Classes in a Nutshell*, which cover the core, graphics, printing, and GUI APIs of the standard Java 2 platform. In addition, a third volume, *Java Enterprise in a Nutshell*, focuses on the Java Enterprise APIs and is of interest to programmers working on server-side or enterprise Java applications.

ISBN 0-596-00253-X
US $29.95
CAN $46.95
90000
9 780596 002534

O'REILLY®

6 36920 00253 6

Visit O'Reilly on the Web at www.oreilly.com

A NUTSHELL® HANDBOOK