



Preface

Most Internet technologies are designed for desktop computers or enterprise servers running on reliable networks with relatively high bandwidth. Handheld wireless devices, on the other hand, have a more constrained computing environment. They tend to have less memory, less powerful CPUs, different input devices, and smaller displays.

Since the mid-1990s, various architectures and protocols have been introduced to deal with these constraints. The Wireless Application Protocol (or WAP), which is a specification developed by the WAP Forum (<http://www.wapforum.org>), takes advantage of several data-handling approaches already in use. Developing wireless applications using WAP technologies is similar to developing Web pages with a markup language (e.g., HTML or XML) because WAP technologies are browser-based.

Another approach to developing wireless applications is to use the Java 2 Platform, Micro Edition (J2ME™). The Java™ programming language already plays an important role in modern programming. With WAP, you can use Java servlets and JavaServer Pages™ to generate Wireless Markup Language (WML) pages dynamically. However, with J2ME, you can now write applications in Java and store them directly on a cell phone. This adds a whole new dimension to wireless programming.

Audience

This book is about programming with J2ME on wireless devices. If you're already familiar with the architecture, you probably noticed that the Connected Limited Device Configuration (CLDC) and the Mobile Information Device Profile (MIDP) classes are not large. Therefore, this book is correspondingly compact in size. The book acts as a quick guide for programmers who are familiar with the Java 2 Standard Edition (J2SE™) and want to get up to speed quickly with the J2ME. We assume that you are familiar with Java programming and have worked with the J2SE classes. In addition, we assume that you are familiar with setting up Java to work under various environments (Windows or Unix platforms), as well as compiling and running Java applications.

The book also serves as a quick reference for Java programmers who are interested in developing wireless software applications. The examples presented throughout the book are a good starting point for working with all the MIDP features, including user interface, networking, and databases. However, we should point out that this book is *not* a rehash of the entire J2SE class library. Several of the classes of `java.io`, `java.lang`, and `java.net` are included in the CLDC and MIDP libraries, but are less bulky than their J2SE counterparts. We assume that you already know how to use these classes, although we have included them in the API reference for completeness.

Contents of This Book

This book is divided into three parts. [Part I](#) gives an overview of the J2ME and includes information about its architectural components: namely, configurations and profiles. [Part I](#) also presents detailed coverage of the CLDC and the MIDP.

[Chapter 1](#)

This chapter introduces the J2ME environment and also explains configurations and profiles. In addition, it shows you how to set up the J2ME Wireless Toolkit to compile, preverify, and run a simple MIDlet using the command line with the Wireless Toolkit emulator.

[Chapter 2](#)

This chapter discusses the CLDC, including its requirements, limitations, and the differences between its classes and the classes of the J2SE. In addition, it looks briefly at the standalone CLDC and KVM distribution.

[Chapter 3](#)

This chapter introduces the requirements, limitations, and classes of the MIDP, as well as introducing MIDlets and their associated Java Application Descriptor (JAD) files.

[Part II](#) contains programming details of the MIDP. It shows you how to program the phone interface, handle events, make network connections, and work with databases.

[Chapter 4](#)

This chapter picks up where [Chapter 3](#) left off, explaining the MIDlet lifecycle methods, the Java application manager, and showing how to use the KToolbar application inside the J2ME Wireless Toolkit to simplify MIDlet development. We also discuss how to deploy MIDlets and include step-by-step instructions on how to download a MIDlet into a Motorola i85s or i50x J2ME-enabled phone.

[Chapter 5](#)

This chapter introduces the MIDP GUI model and its associated classes. In addition, it gives detailed coverage of both the high-level and low-level MIDP GUI APIs.

Chapter 6

This chapter continues the discussion of the MIDP GUI APIs by describing how various events take place surrounding the graphical components and commands. In addition, we cover the `CommandListener` and `ItemStateListener` interfaces, as well as low-level event handling.

Chapter 7

This chapter discusses the Generic Connection Framework provided by the CLDC and shows how to implement an HTTP connection across the Internet, using a MIDlet. The chapter also includes examples of how to send data to CGI scripts and Java servlets across a network. Finally, the chapter briefly discusses wireless session tracking and security for MIDlet data traveling across the airwaves.

Chapter 8

This chapter introduces the concept of data stores, which are simple databases that MIDP applications can use to store persistent data beyond the lifetime of the MIDlet that created them. In addition, the chapter includes a MIDlet that can be used to download stock information from a remote web site.

Chapter 9

This chapter gives a quick introduction to the MIDP implementation on the Palm Connected Organizers, including step-by-step instructions on how to deploy MIDlets to a PalmPilot.

Part III contains several chapters that are quick references for the J2ME CLDC and MIDP APIs. There is also an appendix that contains bibliographic information and URLs to J2ME specifications, white papers, wireless software development kits, and other information that is important to developers.

Conventions Used in This Book

This book uses the following typographical conventions:

A `Constant Width` font is used for:

- Anything that might appear in a Java program, including keywords, data types, constants, method names, objects, variables, class names, and interface names
- All Java code examples
- Attributes that might appear in a manifest or JAD file

An *italic* font is used for:

- New terms where they are defined
- Pathnames, filenames, directory names, and program names (unless the program name is the name of a Java class; then it appears in constant width, like other class names)
- Internet addresses, such as domain names, URLs, and email addresses

A **boldface** font is used for:

- Example lines of Java code to which we wish to draw attention

Acknowledgments

I am deeply grateful to my editor, Robert Eckstein, for all his comments, suggestions, and guidelines throughout the development of this book. I did not know about all the contributions an editor can make to a book until I worked with Bob. Thanks, Bob! Thanks also to the production team at O'Reilly for their hard work on this book.

Special thanks also to Monica Pawlan, Jenny Pratt, Dana Nouri, and Laureen Hudson of the Java Developer Connection (JDC), who either provided comments or edited some of the examples used in this book when they first appeared on the JDC. Also, thanks to the thousands of JDC members who sent in comments and suggestions regarding my articles. Thanks also to the following people who reviewed the contents of this book for accuracy: Ben Griffin, Marc Loy, and Jeff Cunningham.

I would also like to thank my family for their support during my studies, especially my brother, Dr. Mohammad H. Hamdan, for teaching me the value of hard work.

Finally, thanks to my wife, Reema, for her love, support, tolerance, and coffee, and my baby son Yusef, who was born on October 14, 2001, for providing a fun home environment while I finished this book.

Part I: Introducing Java 2 Platform, Micro Edition (J2ME)

Part I is an introduction to the Java 2 Micro Edition (J2ME) and J2ME programming. These chapters will give you an overview of the J2ME, and quickly teach you everything you need to know to get started with J2ME programming.

[Chapter 1](#)

[Chapter 2](#)

[Chapter 3](#)

Chapter 1. Overview of J2ME

This book is about wireless Java programming with the Java 2 Platform, Micro Edition (J2ME). Sun Microsystems, Inc. introduced J2ME at the JavaOne conference in June 1999 as the younger sibling of both the Java 2 Standard Edition (J2SE) and the Java 2 Enterprise Edition (J2EE). At the time, distributed programming was taking the Java developer community by storm, so most of the participants at the show were more interested in what J2EE had to offer. However, over the next two years, developers also realized that there was tremendous value in having small components running Java. Two years later, at the 2001 JavaOne conference, Sun devoted an entire track for individuals seeking to master the once arcane J2ME. Luckily, you don't need to attend JavaOne to learn about J2ME. Instead, this book will help you through the myriad details of understanding J2ME architecture and programming J2ME applications.

In this chapter, we will present an overview of J2ME's primary components, including virtual machines, configurations, and profiles. We'll then present a few short examples of J2ME-enabled applications to whet your appetite and to show you how easy it is to get started with J2ME.

1.1 What Is J2ME?

J2ME is a version of Sun Microsystems' Java that is aimed at the consumer and embedded devices market, which includes electronic commodities such as cellular telephones, pagers, Personal Digital Assistants (PDAs), set-top boxes, and other small devices. Since its release, over 600 companies have joined the development effort, including large corporations such as Palm, Nokia, Motorola, and RIM. However, the direction that J2ME travels is not shrouded in secrecy behind closed corporate doors. Instead, development of J2ME is handled through the Java Community Process (JCP), which allows anyone with an Internet connection to get involved.

J2ME provides a complete set of solutions for creating state-of-the-art networked applications for small devices. It also promises to enable device manufacturers, service providers, and application developers to deploy new applications and services to their customers. However, in doing so, it does not sacrifice some of the founding guidelines of Java, which have become increasingly important these days, namely cross-platform compatibility and security.

1.1.1 A High-Level View

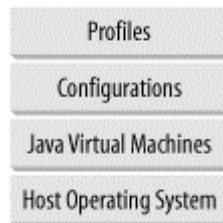
From a high-level view, J2ME defines the following components:

- A series of Java virtual machines, each for use on different types of small devices, each with different requirements
- A group of libraries and APIs that can be run under each of the virtual machines; these are known as *configurations* and *profiles*
- Various tools for deployment and device configuration

The first two components make up the *J2ME runtime environment*. [Figure 1-1](#) provides a relational view of the runtime environment. At its heart is a Java virtual machine, which runs on top of a device's host operating system. Above that is a specific J2ME configuration, which consists of programming libraries that provide

basic functionality based on the resource requirements of the device. On top of the configuration are one or more J2ME profiles, which are additional programming libraries that take advantage of kindred functionalities on similar devices.

Figure 1-1. The high-level architecture of J2ME runtime environment



If you haven't worked with J2ME before, you're probably wondering about the top two layers. It's important to distinguish between a configuration and a profile in the J2ME world, so let's introduce them now.

1.1.2 Configurations

Cellular telephones, pagers, organizers, and other small devices are diverse in form, functionality, and feature. However, they often use similar processors and have similar amounts of memory. For these reasons, the J2ME designers created *configurations*. Configurations define a horizontal grouping of products based on the available memory budget and processing power of each device. Once this information is known, the configuration then outlines the following:

- The Java programming language features supported
- The Java virtual machine features supported
- The basic Java libraries and APIs supported

Currently, there are two standard configurations in the J2ME world: the *Connected Limited Device Configuration* (CLDC) and the *Connected Device Configuration* (CDC). Let's look at the CDC first.

1.1.2.1 The CDC

The CDC is targeted toward powerful devices that are intermittently connected to a network, including set-top boxes, Internet TVs, home appliances, and car navigation systems. The CDC contains a full-featured Java virtual machine, similar to that in use today with the J2SE. The difference lies in the respective devices' memory and display capabilities.

Here are the resource requirements for CDC devices, as given by the official J2ME specifications:^[1]

^[1] The J2ME CDC specifications are located on the Java Community Process web site as JSR-36, which can be found at <http://www.jcp.org/jsr/detail/36.jsp>.

- The device is powered by a 32-bit processor.
- The device has 2 megabytes or more of total memory available for Java. This includes both RAM and flash memory or ROM.

- The device requires the full functionality of the Java 2 "Blue Book" virtual machine.
- The device has connectivity to some kind of network, often with a wireless, intermittent connection and with limited (often 9600 bps or less) bandwidth.
- The device may have a user interface with some degree of sophistication, but a user interface is not mandatory.

1.1.2.2 The CLDC

The second type of configuration is more prevalent in the J2ME world: the CLDC. This configuration specifies a much smaller footprint for consumer and embedded devices than the CDC. The CLDC was first distributed in October 1999 with the idea of creating a "lowest common denominator" Java platform for embedded devices, specifically in terms of networking, I/O, security, and core libraries. Today, some of the devices that you might find powered by the CLDC include mobile cell phones, two-way pagers, personal digital assistants (PDAs), and personal organizers.

Here are the requirements for the J2ME CLDC, again from the official J2ME specifications:*

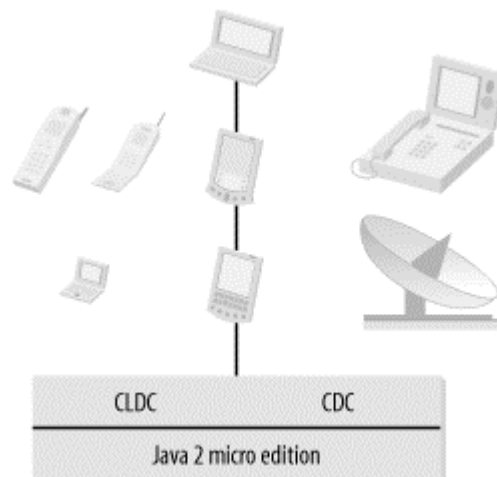
- The device can have between 160 and 512 kilobytes of total memory available for the Java platform, including both RAM and flash memory or ROM.
- The device can have limited power, such as battery-powered operation.
- The device has connectivity to some kind of network, often with a wireless, intermittent connection and with limited (often 9600 bps or less) bandwidth.^[2]

^[2] Note that CLDC stands for *Connected Limited Device Configuration*, not Connectivity-Limited Device Configuration. The difference between the CLDC and the CDC is not in the type or speed of the network connection.

- In addition, the device may have a user interface with some degree of sophistication, but a user interface is not mandatory.

The two products' configurations, along with some of their respective products, are shown in [Figure 1-2](#).

Figure 1-2. J2ME architecture



Note that although the two product groups are supported by different configurations, the line between the two configurations is somewhat blurred. In the future, technological advances will likely make this boundary more and more cloudy. However, for the moment, the important thing to remember is that the boundary between the CLDC and the CDC is defined in terms of the target device's memory budget, battery usage, and the presence or absence of a user interface.

1.1.3 Virtual Machines

As mentioned above, the CLDC and CDC configurations each define their own set of supported features from the Java virtual machine. Consequently, each requires its own Java virtual machine. The CLDC virtual machine is far smaller than the virtual machine required by the CDC, since it supports fewer features. The virtual machine for the CLDC is called the Kilo Virtual Machine (KVM), and the virtual machine for the CDC is called the CVM.

1.1.3.1 The KVM

The KVM is a complete Java runtime environment for small devices. It's a true Java virtual machine as defined by the Java Virtual Machine Specification, except for some specific deviations that are necessary for proper functioning on small devices. It is specifically designed from the ground up for small, resource-constrained devices with a few hundred kilobytes' total memory.

The KVM was originally created as a research project called "Spotless" at the Sun Microsystems Laboratories. The aim of the virtual machine was to implement a Java virtual machine for the resource-constrained Palm Connected Organizer.^[3]

^[3] In fact, early incarnations of the KVM contained several UI libraries based on the "spotless" graphical toolkit.

1.1.3.2 The CVM

The CVM is designed for larger consumer and embedded devices., such as those found with the CDC. It supports all Java 2 Version 1.3 virtual machine features and libraries for items such as security, weak references, JNI, and Remote Method Invocation (RMI). The reference implementation, currently available from Sun Microsystems, runs on Linux and VxWorks. You can download the reference implementation through the J2ME web site at <http://java.sun.com/j2me/>.

Initially, CVM was an acronym for "Compact" Virtual Machine. However, engineers at Sun Microsystems realized that snappy marketers (or poor spellers) may confuse the "compact" in CVM with the K in KVM. So, at present, the C does not stand for anything at all—it is simply known as the CVM.

1.1.4 Profiles

J2ME makes it possible to define Java platforms for vertical product markets by introducing *profiles*. At the implementation level, a profile is a set of APIs that reside on top of a configuration that offers the program access to device-specific capabilities. Following are some examples of profiles that are currently offered through J2ME.

1.1.4.1 The MIDP

The MIDP is designed to be used with the CLDC, and provides a set of APIs for use by mobile devices, such as cellular phones and two-way pagers. The MIDP contains classes for user interface, persistence storage, and networking. It also includes a standardized runtime environment that allows new applications to be "downloaded" to end user devices. Small applications that run under the MIDP are called MIDlets. Since this profile is already released, the vast majority of this book is dedicated to the MIDP.

1.1.4.2 The PDA profile

The PDA profile is based on the CLDC and provides user interface APIs (which are expected to be a subset of the AWT) and data storage APIs for handheld devices. As of this writing, the PDA profile is still in the works and no reference implementation is available yet.

1.1.4.3 The Foundation profile

The Foundation profile extends the APIs provided by the CDC, but it does not provide any user interface APIs. As the name "foundation" implies, this profile is meant to serve as a foundation for other profiles, such as the Personal profile and the RMI profile.

1.1.4.4 The Personal profile

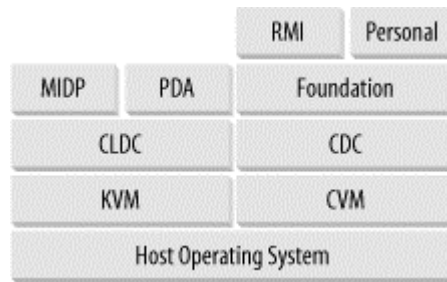
The Personal profile extends the Foundation profile to provide a graphical user interface (GUI) capable of running Java Web applets. Since PersonalJava is being redefined as the Personal profile, it will be backward compatible with PersonalJava 1.1 and 1.2 applications. As of this writing, no reference implementation of the Personal profile is available.

1.1.4.5 The RMI profile

The RMI profile extends the Foundation profile to provide RMI for devices. Since it extends the Foundation profile, the RMI profile is meant to be used with the CDC/Foundation and not the CLDC/MIDP. The RMI profile will be compatible with J2SE RMI API 1.2.x or higher. However, as of this writing, no reference implementation is available yet.

Figure 1-3 shows a global snapshot of current and future J2ME technologies.

Figure 1-3. J2ME environment



1.2 Downloading the J2ME Wireless Toolkit

Now that you know your way around the J2ME landscape, let's get started with J2ME. However, before we can compile and run any J2ME programs, we need to download and install the J2ME Wireless Toolkit. You can obtain the J2ME Wireless Toolkit at the following URL: <http://java.sun.com/products/j2mewtoolkit>.

The version that we use in this book is 1.0.3 beta. It is available for the Microsoft Windows 98/ME and 2000 platforms, as well as Linux and Sun Solaris operating systems. The toolkit requires the presence of at least Version 1.3 of the Java Development Kit (JDK) for the host operating environment.

Once you've downloaded the Wireless Toolkit, double-click on it or execute the resulting binary (depending on your platform) to activate the extraction. This will uncompress the files needed to install the Wireless Toolkit. Note that you may be directed to specify an existing JDK installation on your system. If so, choose the latest stable release of the JDK that you currently have on your system.^[4] In addition, the distribution may also ask you if you would like to install a version of the toolkit that interfaces with Forte™ for Java. If you would like to develop your J2ME applications in the Forte for Java Integrated Development Environment, choose the corresponding option. Be sure that Forte is already installed on your system before doing so.

^[4] Try to use a JDK instead of just a Java Runtime Environment (JRE). It's important that you have the `javac` compiler to create J2ME applications.

In this case, we're going to install the Java Wireless Toolkit on a Windows platform into the directory `C:\j2mewtk`. After the installation is completed, this directory will contain all the required classes and tools to run the MIDP applications. (If the installation program asks you to run the `ktoolbar` program, just ignore it for the moment.) However, we need to do a few more things before we can get started with our examples.

First, we need to add the wireless toolkit binaries to your system path. You can do that on Windows with the following command (again, we've assumed that the Java Wireless Toolkit is installed at `C:\j2mewtk`):

```
SET PATH=%PATH%;C:\j2mewtk\bin
```

If you edit your `C:\AUTOEXEC.BAT` file to add this to the default system path, as shown below, and restart your machine, then you will not have to repeatedly perform this step each time you restart your system.

With Linux and Solaris, the equivalent command is:

```
export PATH=$PATH:install_directory/j2mewtk/bin
```

Once you've added that directory to your system path, you should be able to run the Java Wireless Toolkit tools from any directory on your system. An easy way to test it is to execute the `preverify` command, without any arguments. You should see output similar to the following:

```
C:\> preverify
Usage: PREVERIFY.EXE [options] classnames|dirname ...

where options include:
  -classpath <directories separated by ';'>
                                Directories in which to look for classes
  -d <directory>               Directory in which output is written
  @<filename>                  Read command line arguments from a text file.
```

In order for the toolkit to work properly, you'll need to have the J2SE tools (notably `javac`) available on your system executable path as well. Instructions on how to do this are bundled with the JDK, although it really boils down to adding the binary path of the J2SE binaries to your system path.



If you're familiar with the J2ME Wireless Toolkit already, you're likely wondering why we're not using `KToolbar`. We'll cover `KToolbar` in [Chapter 4](#). In the meantime, it helps to see how J2ME works under the hood.

To compile and run J2ME programs from the command line, enter the following commands. Again, feel free to set these system environment variables on the command line, or edit the `AUTOEXEC.BAT` file (or similar) on your system for convenience.

```
SET J2MEWTK_HOME=C:\j2mewtk
SET MIDPAPI=%J2MEWTK_HOME%\lib\midpapi.zip
SET J2MECLASSPATH=%J2MEWTK_HOME%\wtklib\kenv.zip;
    %J2MEWTK_HOME%\wtklib\kvem.jar;%J2MEWTK_HOME%\wtklib\lime.jar
```

On the Linux and Solaris side, the following could be added to your `.profile` (or equivalent):

```
export J2MEWTK_HOME=/home/qmahmoud/j2mewtk
export MIDPAPI=$J2MEWTK_HOME/lib/midpapi.zip
export J2MECLASSPATH=$J2MEWTK_HOME/wtklib/kenv.zip:
    $J2MEWTK_HOME/wtklib/kvem.jar:$J2MEWTK_HOME/wtklib/lime.jar
```

Note that the final line in either case is really one line; it's been continued here for clarity.

1.3 A Simple Example

The examples that we're going to demonstrate here, and throughout the rest of the book, are called MIDlets. If you've programmed with Java applets or servlets before, then you'll likely recognize the similarities in the "fill-in-the-method" program structure. This first example, `HelloMidlet.java`, shown in [Example 1-1](#), creates a text box and then prints the archetypal "Hello World" in a text box.

Example 1-1. "Hello World"

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

public class HelloMidlet extends MIDlet {

    // The display for this MIDlet
    private Display display;
    // TextBox to display text
    TextBox box = null;

    public HelloMidlet( ) {
    }

    public void startApp( ) {
        display = Display.getDisplay(this);
        box = new TextBox("Simple Example", "Hello World", 20, 0);
        display.setCurrent(box);
    }

    /**
     * Pause is a no-op since there are no background activities or
     * record stores that need to be closed.
     */
    public void pauseApp( ) {
    }

    /**
     * Destroy must cleanup everything not handled by the garbage
     * collector. In this case there is nothing to cleanup.
     */
    public void destroyApp(boolean unconditional) {
    }
}
```

This MIDlet consists of a public class definition that extends the `MIDlet` class found in `javax.microedition.midlet`. This superclass forms the base of all MIDlets in J2ME. Our `HelloMidlet` class contains a constructor, as well as the `startApp()`, `pauseApp()`, and `destroyApp()` methods that have been inherited from the `MIDlet` class. Note that there is no `main()` method in this program. Instead, the `startApp()`, `pauseApp()`, and `destroyApp()` methods are called by the underlying framework to start up the MIDlet, to pause it, or to destroy it.

Let's start off by compiling our program on the command line. Using the command line is a bit more complex than the KToolbar application that comes with the Wireless Toolkit, so in order to simplify it, be sure that you have entered the additional environment variables shown above. However, there are several steps that we need to perform when compiling J2ME applications, and it's important to see each of the steps as they occur.

As you would expect, the program must be saved in a file called `HelloMidlet.java`. However, before you compile it, create a directory called *tmpclasses*. Then use the following command to compile the MIDlet from the command line in Windows:

```
C:\midlets> javac -g:none -d tmpclasses -bootclasspath %MIDPAPI% -  
classpath  
%J2MECLASSPATH% HelloMidlet.java
```

In Linux and Solaris, the command looks like the following:

```
>javac -g:none -d tmpclasses -bootclasspath $MIDPAPI -classpath  
$J2MECLASSPATH  
HelloMidlet.java
```

This command compiles the Java source file without any debugging info, and sets the appropriate boot and J2ME classpaths to ensure that we don't pick up any J2SE classes. The end result of this command is the creation of the `HelloMidlet.class` file in the *tmpclasses* directory.

With the J2SE, a class file was all you needed to run the application. However, all MIDlet classes must be *preverified* before they can be run on a target device. Why is this necessary? Remember that one of the tasks of the standard Java virtual machine (the one that comes with the J2SE) is to perform *bytecode verification*. Bytecode verification is one of the most important steps of the Java security model. It performs such tasks as ensuring that the bytecodes of a Java class (and their operands) are all valid; that the code does not overflow or underflow the VM stack; that local variables are not used before they are initialized; that field, method, and class access control modifiers are respected, and other important tasks. However, most of the bytecode verifier is not included with the KVM due to size constraints. The preverifier ensures that the equivalent security checks still take place.

Before you run the preverifier, create another directory called *classes*. Then, use this command to preverify the `HelloMidlet` class:

```
C:\midlets> preverify -classpath %MIDPAPI%;tmpclasses -d classes  
tmpclasses
```

Or on Solaris and Linux:

```
> preverify -classpath $MIDPAPI:tmpclasses -d classes tmpclasses
```

The resulting output should look something like this:

```
[Output directory for verified classes: classes]
```

This command takes all the classes inside the *tmpclasses* directory (of which `HelloMidlet.class` is the only one) and preverifies them, writing the resulting classes to the *classes* directory. Note that the names of the preverified classes remain exactly the same, which is why we created two separate directories to hold them.



If you received an "Illegal constant pool index" class loading error and you're using JDK 1.4, try using JDK 1.3 until this issue is resolved.

The next step is to compress all the classes in the program (again, we have only one) as well as their resources, into a Java Archive (JAR) file. You can use the J2SE `jar` command to create a JAR file. Make sure you are in the *classes* directory to execute the following command:

```
> jar cvf HelloMidlet.jar HelloMidlet.class
```

The program will compress the `HelloMidlet` class into a JAR file, creating a manifest for it as well.

Note that with the `javac` compiler, you can create MIDlets of practically any size. However, that doesn't guarantee that they will fit on the target device for which you're writing the MIDlet. It would nice if there were a way to check if the target device can handle the MIDlet and run it before it is downloaded. Obviously, if a device can't handle the MIDlet, there is no reason to even attempt a download.

To accomplish this, we need a file that manually specifies some pre-download properties, including the size of the MIDlet and its storage requirements. This can be accomplished by creating a Java Application Descriptor (JAD) file with your favorite text editor. [Example 1-2](#) shows a sample JAD file that we can use. Note that you will need to change the `MIDlet-Jar-Size` entry to correspond to the size of the JAR file that you just created. (In [Chapter 3](#), we will explain the JAD file syntax in more detail.)

Example 1-2. HelloMidlet.jad

```
MIDlet-1: Hello,,HelloMidlet
MIDlet-Name: HelloMidlet
MIDlet-Version: 1.0
MIDlet-Vendor: ORA
MIDlet-Jar-URL: HelloMidlet.jar
MIDlet-Jar-Size: 863
```

Let's save this example JAD file as *HelloMidlet.jad*, again in the *classes* directory that holds the JAR file. Finally, to run this MIDlet, invoke Sun's MIDP emulator to point at the JAD file using the following command:

```
> emulator -Xdescriptor:HelloMidlet.jad
```

If everything worked correctly, you should see a phone similar to [Figure 1-4](#), although the display may be different. Here, the `HelloMidlet` is running in the default phone that comes with the Java Wireless Toolkit. If you click on the MIDlet on the menu (use the directional arrow pad to move the cursor and the button in the middle to select), and instruct it to "Launch" using the soft button on the lower right, you should see output similar to [Figure 1-4](#). Congratulations! You just created your first Java MIDlet!

Figure 1-4. HelloMidlet



The gist of this program is in the `startApp()` method. Here, we obtain the current display that the device uses, then create a text box with the words "Hello World" inside of it. Finally, we show the text box on the current display. Don't worry if you don't understand these objects yet; the architecture of MIDlets will become clearer as we move through the book.

1.3.1 A Login MIDlet

Let's move to a more advanced MIDlet. [Example 1-3](#) shows a MIDlet with a hypothetical login screen that prompts the user to log in. If the login is incorrect, the program will repeatedly ask the user to try again.

Example 1-3. A login MIDlet

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

public class LoginMidlet extends MIDlet implements CommandListener {
    private Display display;
    private TextField userName;
    private TextField password;
    private Form form;
    private Command cancel;
    private Command login;

    public LoginMidlet() {
        userName = new TextField("LoginID:", "", 10, TextField.ANY);
        password = new TextField("Password:", "", 10,
TextField.PASSWORD);
        form = new Form("Sign in");
        cancel = new Command("Cancel", Command.CANCEL, 2);
```

```

        login = new Command("Login", Command.OK, 2);
    }

    public void startApp( ) {
        display = Display.getDisplay(this);
        form.append(userName);
        form.append(password);
        form.addCommand(cancel);
        form.addCommand(login);
        form.setCommandListener(this);
        display.setCurrent(form);
    }

    public void pauseApp( ) {
    }

    public void destroyApp(boolean unconditional) {
        notifyDestroyed( );
    }

    public void validateUser(String name, String password) {
        if (name.equals("qm") && password.equals("j2")) {
            menu( );
        } else {
            tryAgain( );
        }
    }

    public void menu( ) {
        List services = new List("Choose one", Choice.EXCLUSIVE);
        services.append("Check Mail", null);
        services.append("Compose", null);
        services.append("Addresses", null);
        services.append("Options", null);
        services.append("Sign Out", null);
        display.setCurrent(services);
    }

    public void tryAgain( ) {
        Alert error = new Alert("Login Incorrect", "Please try again",
null,
        AlertType.ERROR);
        error.setTimeout(Alert.FOREVER);
        userName.setString("");
        password.setString("");
        display.setCurrent(error, form);
    }

    public void commandAction(Command c, Displayable d) {
        String label = c.getLabel( );
        if(label.equals("Cancel")) {
            destroyApp(true);
        } else if(label.equals("Login")) {
            validateUser(userName.getString(), password.getString( ));
        }
    }
}

```

Again, don't worry if you can't understand the entire program at this point; this example is just meant to give you a flavor of MIDP programming and some sample

applications to compile and run. [Chapter 5](#) and [Chapter 6](#) will explain the GUI classes (such as `Display`, `Form`, and `TextField`), as well as the event-handling classes (such as `Command`) in much more detail.

That being said, let's present a beginner's overview of how this MIDlet works. As in the previous example, `LoginMidlet` extends the `MIDlet` abstract class. It also implements the `CommandListener` interface by providing an implementation for the `commandAction( )` method. In this method, there are two commands: `Login` and `Cancel`. The label of the command is checked: if it is `Cancel`, the `LoginMidlet` is destroyed, and if it is `Login`, then the username and passwords are validated.

In the `LoginMidlet`'s constructor, a `Form` object, two `TextField` objects, and two `Command` objects are created. The `TextField` and `Command` objects are added to the form in the `startApp( )` method. In addition, `pauseApp( )` and `destroyApp( )` perform minimal tasks.

Here is how the program operates: if the `Login` command is given, the application calls the `validateUser( )` method to validate the username and password. If they are valid (in this case, they are hardcoded into the program for simplicity), then the `menu( )` method is called to simulate a list of "useful services." Otherwise, the `tryAgain( )` is called to display an error message and to allow the user to reenter their name and password.

If you are using the command line to compile and execute, save this file named *LoginMidlet.java*, make sure that you have a *classes* and a *tmpclasses* directory, and use `javac`:

```
C:\midlets> javac -g:none -d tmpclasses -bootclasspath %MIDPAPI% -
classpath
    %J2MECLASSPATH% LoginMidlet.java
```

If you are using Solaris or Linux, the command becomes:

```
>javac -g:none -d tmpclasses -bootclasspath $MIDPAPI -classpath
$J2MECLASSPATH
    LoginMidlet.java
```

Next, remember that we must preverify the resulting class:

```
C:\midlets> preverify -classpath %MIDPAPI%;tmpclasses -d classes
tmpclasses
```

or

```
> preverify -classpath $MIDPAPI:tmpclasses -d classes tmpclasses
```

Again, the preverified class is saved to the *classes* subdirectory in the current directory. Next, compress the resulting class into a JAR file:

```
jar cvf LoginMidlet.jar LoginMidlet.class
```

And finally, create a JAD file that describes the resulting JAR file in detail, as shown in [Example 1-4](#).

Example 1-4. LoginMidlet.jad

```
MIDlet-1: Login,,LoginMidlet
MIDlet-Name: LoginMidlet
MIDlet-Version: 1.0
MIDlet-Vendor: ORA
MIDlet-Jar-URL: LoginMidlet.jar
MIDlet-Jar-Size: 1786
```

Again, don't forget to change the size of the JAR file to match the size of the *LoginMidlet.jar* file after you create it.

At this point, the MIDlet can be run as in the previous example, using the MIDP emulator of the Java Wireless Toolkit, with the following command:

```
emulator -Xdescriptor:LoginMidlet.jad
```

In addition, the MIDlet can be run with any other emulator you may have available. For example, to whet your appetite, [Figure 1-5](#) shows the LoginMidlet running on the Motorola i85s emulator (the i85s is a J2ME-enabled cell phone available from Motorola and Nextel).

Figure 1-5. LoginMidlet running in the Motorola i85s emulator (cropped)



1.3.2 Working with the Emulator

Note that the objects represented by the `Command` class are shown above the two "soft buttons" on the phone (the buttons with the black circles). If a soft button below the command is pressed, the command immediately above it is executed. Here, if the user enters the correct username and matching password and presses the Login button, the menu of services will be displayed. Otherwise, the alert will be displayed and the user can try again.

Also, you might be caught off guard the first time you try to enter text with your computer keyboard. It doesn't work! That's because you must use the input keys on the phone to enter the text. In this case, to enter the letter "G", press the number "4." To enter the letter "K", press the number "5" twice. Note how each time you press a

numeral, the system "cycles" through the letter corresponding to that number. To move down to entering text for the password, use the down arrow.

Well, that's it! You've just created two professional MIDlets using J2ME! In the next two chapters, we're going to take a much closer look at the CLDC and the MIDP, two exciting new areas of wireless Java development.

Chapter 2. The Connected Limited Device Configuration (CLDC)

The Connected Limited Device Configuration (CLDC) defines a standard, minimum-footprint Java platform for small, resource-constrained devices. As we mentioned in [Chapter 1](#), the CLDC was designed as a lowest common denominator of Java that can be applicable to a wide variety of devices. However, features specific to a certain vertical market, such as cell phones or pagers, are not found in the CLDC but are instead defined in profiles that sit above it. Configurations primarily target devices with similar amounts of memory and processing power.

This leads to a very important point about the CLDC: there are no optional features. Everything that the CLDC provides is usable on the devices that support it. After all, the primary goal of the CLDC is to ensure portability and interoperability between applications running on various kinds of resource-constrained devices, which is the main objective of programming in Java. In this chapter, we discuss the CLDC and its virtual machine, the KVM, in detail.

2.1 Examining the CLDC in Detail

Let's start off with some specifics. According to the specification, the devices targeted by the CLDC have the following characteristics:

160 KB to 512 KB of total memory

At a minimum, a CLDC device should have 128 KB of non-volatile memory for the Java VM and the CLDC libraries, and at least 32 KB of volatile memory for the VM to use at runtime, independent of any applications.

16-bit or 32-bit processor with at least 25 Mhz speed

These types of processors are pretty typical in today's handheld devices.

Connectivity to some kind of networking

With CLDC, this is often a two-way wireless connection with limited bandwidth.

Low power consumption

CLDC devices often operate under battery power. Hence, they have very low power consumption.

Devices that fit these characteristics come in all shapes and sizes. Cell phones and pagers immediately come to mind, but one could also install Java on bar code scanners, video and audio equipment, navigation systems, and other wireless devices yet to come. In fact, as the nature of these devices changes, you can expect that the base specifications for the CLDC will change as well.

Given the constraints listed above, the CLDC currently provides the following functionality to its devices:

- A subset of Java language and virtual machine features
- A subset of core Java libraries (`java.lang` and `java.util`)
- Basic input/output (`java.io`)
- Basic networking support (`javax.microedition.io`)
- Security

Note, however, that the CLDC does not address application life cycle management, user interfaces, event handling, or the interaction between the user and the application. Again, these features fall into the domain of profiles, such as the MIDP, which are implemented on top of the CLDC and add to its functionality.

2.1.1 What's Different About the Java Virtual Machine?

We mentioned that the CLDC does not have any optional features. As you might expect, this means that a number of features have been eliminated from Java virtual machines that support the CLDC, either because they are too expensive (in terms of memory or processing capability) to implement, or because their presence would impose security problems. Therefore, if you're new to programming with the CLDC, you should be aware of the following limitations in CLDC VMs:

No floating point support

The CLDC does not support floating point numbers; therefore, CLDC-based applications cannot use floating point types such as `float` or `double`. This decision was made because most CLDC target devices do not have floating point support in their underlying hardware.

No finalization

The CLDC API currently does not include the `Object.finalize( )` method; you cannot perform final cleanup operations on object data—such as closing resources—before an object is garbage-collected.

Limited error handling

Runtime errors are handled in an implementation-specific manner. The CLDC defines only three error classes: `java.lang.Error`, `java.lang.OutOfMemoryError`, and `java.lang.VirtualMachineError`. Non-runtime errors are handled in a device-dependent manner that often involves terminating the application or even resetting the device.

No Java Native Interface (JNI)

A Java virtual machine supporting the CLDC does not implement the JNI. There are actually two good reasons for this: security, and the fact that implementing JNI is expensive, given the memory constraints of CLDC target devices.

No user-defined class loaders

A Java virtual machine supporting the CLDC must have a built-in class loader that cannot be overridden or replaced by the user. This is for security reasons.

No support for reflection

CLDC applications do not have the ability to use the reflection APIs on their objects or classes. Because reflection is not supported, there is also no support for object serialization or RMI.

No thread groups or daemon threads

While a Java virtual machine that supports the CLDC will implement multithreading, it cannot support thread groups or daemon threads. If you want to perform thread operations for groups of threads, use the collection objects to store the thread objects at the application level.

No weak references

No application built on a Java virtual machine supporting the CLDC can require weak references.

2.1.2 The KVM

The KVM, which was introduced in the previous chapter, is a complete Java runtime environment for small devices. It is a true Java virtual machine as defined by the Java Virtual Machine Specification, except for some deviations that are necessary for proper functioning on small devices. The KVM was specifically designed for small, resource-constrained devices that have only a few hundred kilobytes total memory.

The J2ME white paper^[1] describes the KVM as:

^[1] See also the KVM white paper, located at <http://java.sun.com/products/cldc/wp/KVMwp.pdf>, for much more detail on the KVM.

- Designed for both 16-bit and 32-bit CISC or RISC processors and clocked at processors as low as 25 Mhz
- Small, with a static memory footprint of 50 to 80 KB
- Highly portable, modular, and customizable
- As complete and fast as possible, without sacrificing the other design goals listed above

The KVM was derived from a research project called Spotless at Sun Microsystems Laboratories. The aim of the project was to implement a Java system for the Palm Connected Organizer.^[2] The KVM is written in the C programming language (using about 24,000 lines of code), so it can be easily ported to various platforms for which a C-language compiler is available. Finally, like a regular JVM, the KVM can load classes from a class path directory as well as from a JAR file.

^[2] If you attended JavaOne 1999, you'll remember that this was a major attraction. They even held a contest to see who could design the best KVM application.

2.1.2.1 Class Verification

In the J2SE Java virtual machine, the class verifier is responsible for rejecting invalid class files at runtime. A JVM supporting CLDC must be able to reject invalid class files as well. The class verification process, however, is expensive and time-consuming: it typically takes anywhere from 35 to 110 KB of runtime memory. Since the target size of the KVM is 50 to 80 KB of memory, including a class verifier inside it would violate its size constraints.

The KVM designers decided to move most of the verification work off the device and onto the desktop, where the class files are compiled, or onto the server machine, from which applications are downloaded. This step (off-device class verification) is referred to as *preverification*; that's why we had to run the `preverify` command on the examples in [Chapter 1](#). Once the preverification is completed, the resulting class files often include extra information to ensure that the runtime verifier can perform its job with only minimal effort and memory. (That's why the preverified version of the `LoginMidlet.class` in [Chapter 1](#) is slightly larger than the raw class generated by the `javac` compiler.)

The additional output of the preverification process is the addition of a stack map attribute that maps out critical areas of a class. This additional attribute is used by the runtime verifier to pinpoint critical areas inside the class that must be checked. Also, the preverifier will inline all subroutines in the bytecodes of the class file to prevent any problems at runtime. Don't worry, however. Even with the additional information, the preverified class files can still work with a regular Java runtime verifier.

With the help of the preverification, the CLDC device is only responsible for running a quick scan on the preverified class file to ensure that it was verified and does not contain any illegal instructions. This cuts down significantly on the amount of memory needed for the runtime verifier: only 100 bytes or so.

2.1.2.2 Security

The CLDC security model is more strict than what you're likely used to with the J2SE. This new security model is primarily concerned with two areas:

Virtual machine-level security

An application executed by the KVM must not be able to harm the device in which it is running. This is guaranteed by the class verifier, which ensures that the class bytecodes cannot contain references to invalid memory locations. It

also ensures that the classes loaded cannot execute in a way that is not allowed by the Java Virtual Machine Specification. As we mentioned, class verification for the CLDC/KVM is a two-step process: off-device preverification in conjunction with a minimal in-device verification. In addition, native methods cannot be invoked at runtime.

Application-level security

Unlike the J2SE, the CLDC/KVM combination does not allow the customization of a security manager. A JVM supporting CLDC provides a simple sandbox security model that enforces security by ensuring that applications run in a closed environment, and that applications may only call classes supported by the device.

2.1.3 What's Different About the Core Java Libraries?

The first thing that you'll probably notice when working with the CLDC is that only a bare minimum of Java APIs have been included. The reason for this is obvious if you download the Java 2 SDK: the standard edition APIs require close to 20 megabytes of memory! This is memory that most small devices simply do not have. Hence, one of the primary goals in designing the core CLDC libraries was to boil the J2SE APIs off into a minimum set of libraries that could still be used for meaningful application and profile development.

With that in mind, it's helpful to think of the CLDC library APIs as divided into two categories: classes that are a subset of the J2SE APIs and new classes that are specific to the CLDC. Let's look at the former group first.

2.1.3.1 Classes inherited from J2SE

The CLDC uses only thirty-seven classes from the J2SE platform. These classes come from the `java.lang`, `java.io`, and `java.util` packages, which are derived from JDK 1.2 APIs. Note that according to the J2ME specification, "Each class that has the same name and package name as a J2SE class must be identical to, or a subset of, the corresponding J2SE class. The semantics of the classes and methods cannot be changed, and the classes cannot add any `public` or `protected` methods or fields that are not available in the corresponding J2SE class libraries." In other words, you cannot add, but you can take away. And many classes have functionality taken away.

The inherited classes and interfaces (not including exceptions) from the J2SE platform are shown in [Table 2-1](#).

Table 2-1. Inherited, non-exceptional classes	
Package	Classes
<code>java.lang</code>	<code>Boolean</code> , <code>Byte</code> , <code>Character</code> , <code>Class</code> , <code>Integer</code> , <code>Long</code> , <code>Math</code> , <code>Object</code> , <code>Runnable</code> , <code>Runtime</code> , <code>Short</code> , <code>String</code> , <code>StringBuffer</code> , <code>System</code> , <code>Thread</code> , <code>Throwable</code>
<code>java.io</code>	<code>ByteArrayInputStream</code> , <code>ByteArrayOutputStream</code> , <code>DataInput</code> , <code>DataOutput</code> , <code>DataInputStream</code> , <code>DataOutputStream</code> , <code>InputStream</code> , <code>OutputStream</code> , <code>InputStreamReader</code> , <code>OutputStreamWriter</code> , <code>PrintStream</code> , <code>Reader</code> , <code>Writer</code>
<code>java.util</code>	<code>Calendar</code> , <code>Date</code> , <code>Enumeration</code> , <code>Hashtable</code> , <code>Random</code> , <code>Stack</code> , <code>TimeZone</code> , <code>Vector</code>

Because all inherited classes must throw precisely the same exceptions as regular J2SE classes, the following 29 exception and error classes shown in [Table 2-2](#) also derive from the J2SE APIs.

Table 2-2. Inherited exception and error classes	
Package	Class
java.lang	ArithmeticException, ArrayIndexOutOfBoundsException, ArrayStoreException, ClassCastException, ClassNotFoundException, Error, Exception, IllegalAccessException, IllegalArgumentException, IllegalMonitorStateException, IllegalThreadStateException, IndexOutOfBoundsException, InstantiationException, InterruptedException, OutOfMemoryException, NegativeArraySizeException, NumberFormatException, NullPointerException, RuntimeException, SecurityException, StringIndexOutOfBoundsException, VirtualMachineError
java.io	EOFException, IOException, InterruptedIOException, UnsupportedEncodingException, UTFDataFormatException
java.util	EmptyStackException, NoSuchElementException

When programming with the CLDC, there are many internal modifications to the J2SE classes you're used to. Here are some of the more common classes that may cause problems.

2.1.3.2 String and StringBuffer

The following methods have been removed from the ubiquitous `java.lang.String` class, either because they refer to floating-point data types or because their presence is redundant:

```
public void valueOf(float f)
public void valueOf(double d)
public int compareToIgnoreCase(String str)
public boolean equalsIgnoreCase(String anotherStr)
public static copyValueOf(char[] data)
public static String copyValueOf(char[] data, int offset,
    int count)
public String intern( )
public int lastIndexOf(String str)
public int lastIndexOf(String str, int fromIndex)
public boolean regionMatches(int toffset, String other,
    int ooffset, int len)
public String toLowerCase(java.util.Locale locale)
public String toUpperCase(java.util.Locale locale)
```

For the same reasons, the following methods have been eliminated from the `java.lang.StringBuffer` class.

```
public StringBuffer append(float f)
public StringBuffer append(double d)
public StringBuffer insert(int offset, float f)
public StringBuffer insert(int offset, double d)
public StringBuffer insert(int index, char[] str,
    int offset, int len)
public StringBuffer replace(int start, int end, String str)
public String substring(int start)
public String substring(int start, int end)
```

2.1.3.3 Runtime

The `java.lang.Runtime` class has eliminated most of its methods for the CLDC. Here, only the following subset of methods is now available:

```
public void exit(int status);
public native long freeMemory( );
public native void gc( );
public static Runtime getRuntime( );
public native long totalMemory( );
```

2.1.3.4 System

In addition, the `java.lang.System` class only has the following fields and methods available to it:

```
public static final PrintStream err;
public static final PrintStream out;
public static native void arraycopy(Object src,
    int src_position, Object dst, int dst_position, int length);
public static native long currentTimeMillis( );
public static void exit(int status);
public static void gc( );
public static String
getProperty(String key);
public static native int identityHashCode(Object x);
```

2.1.3.5 Math

Finally, as you might expect, the `java.lang.Math` class has eliminated all methods dealing with complex floating-point operations (which was the vast majority of methods in that class), and now only has the following methods:

```
public static int abs(int a);
public static long abs(long a);
public static int max(int a, int b);
public static long max(long a, long b);
public static int min(int a, int b);
public static long min(long a, long b);
```

In many cases, the absence of these methods are only a minor inconvenience and suitable workarounds can be used. J2ME functionalities that require the use of floating-point values, however, may have to expand their floating-point values to integers with an implied decimal point and improvise with the more limited set of integer operations.

2.1.4 What's Different About I/O and Networking?

Recall that the J2SE provides the `java.io` and `java.net` packages for I/O and network connectivity. The CLDC inherits some of the classes in the `java.io` package. However, the major difference is that it does not inherit classes related to *file* I/O. For example, the popular `FileInputStream` and `FileOutputStream` classes are not present. In addition, the `FileReader` and `FileWriter` classes are not offered for

reading and writing text data. This is because not all CLDC devices support the concept of a filesystem.

As for the `java.net` package, the J2SE provides several classes for network connectivity. However, none of these classes have been inherited because not all devices require TCP/IP or UDP/IP. (Some devices may not even have an IP stack.) Instead, the CLDC expert group decided to define a more generic set of classes for J2ME I/O and network connectivity. These classes are known as the *Generic Connection Framework*, and are found in the `javax.microedition.io` package.

2.1.4.1 The Generic Connection Framework

The Generic Connection Framework is a platform-independent framework that provides its functionality without any dependence on the specific features of a device. In fact, this framework is so generic that it doesn't implement any of the I/O or network connectivity interfaces; rather, the profile above it provides such implementation.

Here's a quick rundown of how the Generic Connection Framework works: all connections are created using the static `open()` method from the `factory Connector` class. If successful, this method returns an object that implements one of the generic connection interfaces for the host device. If you're a J2SE programmer, this will be much different than what you're used to. However, it will also be much easier. To give you a taste of what this is like, here are some example connections from the J2ME specification that you might request from a CLDC application and the appropriate syntax to implement them:

HTTP connection

```
Connector.open("http://www.ora.com:port");
```

Socket connection

```
Connector.open("socket://www.ora.com:port");
```

Communication with a port

```
Connector.open("comm:0;baudrate=9600");
```

The goal of the above syntax is to isolate any differences in the protocol that you're attempting to connect with into a simple string. This way, most of the application's code remains the same, regardless of the protocol you use. The Generic Connection Framework is discussed in more detail in [Chapter 7](#).

2.1.5 Differences with Property Support in the CLDC

Virtual machines that support the CLDC, such as the KVM, do not implement the `java.util.Properties` class, which follows from the lack of filesystem functionality that we mentioned above. However, four system properties are supported for each J2ME/CLDC device, and can be accessed by using the method

`System.getProperty(String key)`. The four properties available are described in [Table 2-3](#).

Table 2-3. System properties		
System property	Description	Default value
<code>microedition.platform</code>	Name of the host platform or device	<code>null</code>
<code>microedition.encoding</code>	Default character encoding	<code>"ISO8859_1"</code>
<code>microedition.configuration</code>	Name and version of the support configuration	<code>"CLDC-1.0"</code>
<code>microedition.profiles</code>	Name of the supported profiles	<code>null</code>

There's very little to say here, except that you can use these properties to ensure that you're indeed on a CLDC device that supports the proper encoding and profiles for your application. The MIDP profile defines some additional properties, which we will discuss in [Chapter 3](#).

2.2 Using the Standalone CLDC and KVM

If you want to experiment with the raw KVM and CLDC classes, you can download the standalone CLDC and KVM. As of this writing, the latest edition of the CLDC itself is version 1.0.2. The CLDC 1.0.2 contains an updated version of the KVM. The KVM code has been rewritten to improve performance and includes a faster bytecode interpreter, better garbage collection, Java-level debugging APIs, preverifier improvements, and several bug fixes. If you wish to download the standalone CLDC and KVM, you can find it at the following address: <http://java.sun.com/products/kvm>.



Note that this is different than the J2ME Wireless Toolkit that we used in [Chapter 1](#). This distribution does not contain any MIDP classes, nor does it contain a MIDP emulator. Hence, it will only execute programs that adhere to the base CLDC specification and not any MIDP functionality. If you are solely interested in writing applications for the MIDP, you can just read through this section without taking any action.

This distribution contains KVM implementations for Windows, Solaris, and Linux operating systems, as well as the CLDC classes that can be used to compile and run applications. After downloading and uncompressing the distribution, you should have a series of directories, as shown in [Table 2-4](#).

Table 2-4. CLDC/KVM directories	
Directory	Description
<code>api</code>	The Java classes and source code for the CLDC
<code>bin</code>	Binaries for each of the target platforms
<code>build</code>	Utility that builds directories and makefiles for each target platform
<code>docs</code>	PDF documentation, as well as compressed javadocs
<code>jam</code>	Java Application Manager, which can be used to dynamically download classes into the KVM
<code>kvm</code>	Source and build files pertaining to the KVM
<code>samples</code>	Sample code that can be used with the CLDC

tools	Source for the various tools used with the CLDC
-------	---

Feel free to look through the *api* directory to see what you have. It's not much, compared to the J2SE. In any case, there are some interesting things that we can show you with the KVM and the CLDC in this distribution. First, create a simple program that can be run with the CLDC as follows:

```
public class CLDCTest {
    public static void main(String[] args) {
        System.out.println("Hello CLDC!");
    }
}
```

Then, try compiling the program with the standard `javac` compiler. In the example below, we use a command line, similar to that in the first chapter, in order to ensure that only the CLDC classes are used. Note that the subsequent series of commands assumes that you are in the base directory of the CLDC/KVM distribution:

```
javac -bootclasspath api/classes.zip CLDCTest.java
```

Remember that we must preverify our resulting class before running it with the KVM, like we did in [Chapter 1](#). You can do so with the following command:

```
bin/[targetOS]/preverify -classpath api/classes.zip:. CLDCTest
```

As before, this should create a separate directory, here called *output*, where the preverified class has been stored. You can now run this class with the KVM, using the following command:

```
bin/[targetOS]/kvm -classpath api/classes.zip:output CLDCTest
Hello CLDC!
```

Next, try modifying the source code so that it adds a declaration of the float variable:

```
public class CLDCTest {

    float f;

    public static void main(String[] args) {
        System.out.println("Hello CLDC!");
    }
}
```

And again, try recompiling it and preverifying it with the above commands. If you try running the resulting program, the KVM will flag the inclusion of a floating-point field in the class as an error:

```
ALERT: Bad field signature
```

Why didn't the compiler flag the use of the floating-point variable as an error? Remember that you're using the `javac` compiler from J2SE to compile your J2ME programs, and that compiler is all too familiar with the use of floating-point variables. Hence, it will assume that the primitive data types that it knows about are fine for use

with whatever JVM is on the other side of the compilation. In addition, the preverifier will not search for floating-point variables because its job (at least, on the desktop side) is to look for security issues within classes, not to hunt down invalid primitive data types. (Remember we mentioned earlier that preverified class files must work under the regular J2SE.) Hence, the KVM itself has to tell us that one of our fields is not supported in the virtual machine, which it does by scanning through the class files before executing them. There's an important lesson here: just because the compiler and preverifier successfully translated a source file to a class with only the CLDC classes on its bootclasspath doesn't mean that it will still run. You should always test it with the KVM as well, to see if the code has any VM issues.

That's not to say that if we used a method that is no longer in the CLDC classes, the compiler wouldn't notice. For example, assume that we modified our code to be the following:

```
public class CLDCTest {

    static String s = "Hello CLDC!";
    static int r = s.compareToIgnoreCase("HELLO CLDC!");

    public static void main(String[] args) {
        System.out.println(s + ":" + r);
    }
}
```

This yields the following compiler error:

```
CLDCTest.java:4: cannot resolve symbol
symbol:   method compareToIgnoreCase(java.lang.String)
location: class java.lang.String
```

Here, the compiler flagged an error because the `String` class that was located on its bootclasspath does not contain the method in question, `compareToIgnoreCase()`. As we mentioned earlier in the chapter, this method has been omitted in the CLDC subset of `java.lang.String`.

2.3 CLDC Next Generation

Finally, let's briefly mention the CLDC Next Generation (NG). The CLDC NG is a specification that is currently in development and that aims to define a revised version of the CLDC. The goal of the CLDC NG is to make the CLDC more compliant with the Java language and virtual machine specifications by reintroducing features such as floating-point support and improved error-handling capabilities.

Some other goals of the CLDC NG will be to:

- Maintain backward compatibility with CLDC 1.0.
- Maintain small footprint (limit API growth).
- Continue focus on small, resource-constrained, connected devices.
- Investigate the possibility of adding a minimal security manager.

Note, however, that not many new APIs will be reintroduced to the CLDC with this revision. Devices that require significantly more complete Java libraries should use the Connected Device Configuration (CDC) instead. You can follow the progress of the CLDC NG at the Java Community Process web site at: <http://www.jcp.org/>.

Chapter 3. The Mobile InformationDevice Profile (MIDP)

The Mobile Information Device Profile (MIDP) is built on top of the CLDC, and defines an open application development environment for what Sun calls *Mobile Information Devices* (MIDs). In simpler terms, MIDP is the J2ME profile that is used for wireless devices, such as mobile phones and pagers. This chapter expands on the previous chapter by introducing some of the fundamental concepts of MIDP and offering programming guidelines that are used throughout the remainder of this book.

As we mentioned in [Chapter 1](#), the MIDP is governed by the Java Community Process. The MIDP is JSR 37, which is part of the Java Community Process. Like the CLDC, the MIDP is an ever-changing standard that actively solicits input from corporations and the general programming community. You can find more information on the MIDP at the following URL: <http://java.sun.com/products/midp>.

3.1 Mobile Information Devices

Again, let's start off with some specifics. The MIDP standard defines a MID as a device with the following minimum characteristics:

Display

A screen size of at least 96 x 54 pixels with at least a 1-bit display depth

Input

A one-handed keyboard, two-handed keyboard, or touch screen

Memory

32 KB of volatile memory for the Java runtime (heap); 128 KB of non-volatile memory for the MIDP components; and 8 KB of non-volatile memory for application-created persistent data

Networking

A two-way intermittent connection, usually wireless, with limited bandwidth

Because the MIDP is built on top of the CLDC, it addresses the following areas that are omitted by the CLDC:

Application Life Cycle Management

The MIDP includes the `javax.microedition.midlet` package, which contains classes and methods for starting, pausing, and destroying applications in the host environment.

User Interface and Events

The MIDP also provides the `javax.microedition.lcdui` packages, which include classes and interfaces for creating GUI components for applications.

Network Connectivity

The MIDP extends the `ContentConnection` interface of the Generic Connection Framework by providing the `HttpConnection` interface, as well as a subset implementation of the HTTP protocol.

Storing Data on Device

The MIDP also provides the `javax.microedition.rms` package, which implements a record-based database management system. This provides applications with the capability to store data on the device.

The MIDP has received wide corporate backing, from companies such as AOL, DDI, Ericsson, Fujitsu, Hitachi, Matsushita, Mitsubishi, Motorola, NEC, Nokia, NTT DoCoMo, Palm, Research in Motion (RIM), Samsung, Sharp, Siemens, Sony, Sprint, and Symbian. In the second quarter of 2001, Motorola released the first MIDP-enabled cellular phones, the i50x and the i85s.^[1] Over the next year or two, you'll likely see an impressive amount of MIDP-enabled devices reach the market.

^[1] The service for the i50x and i85s phones is provided in the United States and Canada by Nextel, Inc.

3.1.1 Class Additions

The MIDP adds the following packages to those available through the CLDC, as shown in [Table 3-1](#).

Table 3-1. New packages in the MIDP	
Package	Description
<code>javax.microedition.lcdui</code>	Graphical interface components and events
<code>javax.microedition.midlet</code>	Application life cycle
<code>javax.microedition.rms</code>	Record storage

Here are the classes that are included with each of the new packages. The first package, `javax.microedition.lcdui`, contains interfaces and classes, listed in [Table 3-2](#), that are used to build graphical interfaces on the limited displays of CLDC devices. These classes are discussed in detail in [Chapter 5](#) and [Chapter 6](#).

Classes and interfaces in the `javax.microedition.lcdui` package

Name	Type
Choice	Interface

CommandListener	Interface
ItemStateListener	Interface
Alert	Class
AlertType	Class
Canvas	Class
ChoiceGroup	Class
Command	Class
DateField	Class
Display	Class
Displayable	Class
Font	Class
Form	Class
Gauge	Class
Graphics	Class
Image	Class
ImageItem	Class
Item	Class
List	Class
Screen	Class
StringItem	Class
TextBox	Class
TextField	Class
Ticker	Class

The next package, `javax.microedition.midlet` (see [\[click here\]](#)), adds only one class that serves as the base class for all MIDlets. This class can only throw one exception as well, which notifies listeners of a state change in the MIDlet. This class is discussed in detail in [Chapter 4](#).

Class and exception in the `javax.microedition.midlet` package

Name	Type
MIDlet	Class
MIDletStateChangeException	Exception

Finally, the `javax.microedition.rms` package provides four interfaces, one class, and five exceptions (see [Table 3-4](#)) for performing persistent data storage on MIDP devices. The four interfaces allow you to create implementing classes that customize how the record store compares, enumerates through, filters, and handles events that occur with data records. These classes are discussed in detail in [Chapter 8](#).

The classes, interfaces, and exceptions in the `javax.microedition.rms` package

Name	Type
RecordComparator	Interface
RecordEnumeration	Interface
RecordFilter	Interface
RecordListener	Interface

RecordStore	Class
InvalidRecordIDException	Exception
RecordStoreException	Exception
RecordStoreFullException	Exception
RecordStoreNotFoundException	Exception
RecordStoreNotOpenException	Exception

In addition to these packages, the MIDP also adds two classes and one exception to those classes in the `java.lang` and `java.util` packages of the CLDC. These classes are similar to those found in the Java SDK 1.3.

- `java.lang.IllegalStateException` (exception)
- `java.util.Timer` (class)
- `java.util.TimerTask` (class)

As you can see, there aren't many classes in the MIDP. However, that's not unexpected, given that we need to fit MIDP programs in such a limited space. But don't worry. We'll discuss each of the new classes, interfaces, and exceptions of the MIDP as we progress through the remainder of the book.

3.1.2 System Properties

The MIDP defines two additional property values (in addition to the eight in the previous chapter) that can be retrieved using the `java.lang.System.getProperty()` method. These are shown in [Table 3-2](#):

Table 3-2. System properties defined by the MIDP	
System property	Description
<code>microedition.local</code>	The current locale of the device (default: null)
<code>microedition.profiles</code>	Must contain at least "MIDP-1.0"

The `microedition.local` property consists of the language and country code separated by a dash "-". For example, "en-CA" for English Canada and "en-US" for English USA. Note that the language code must be lowercase, and the country code must be uppercase.

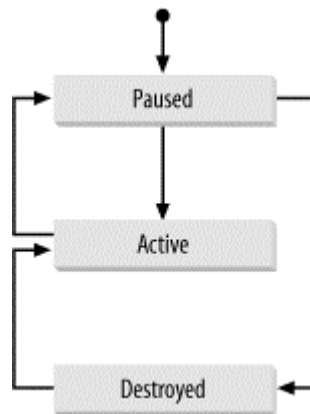
3.2 More About MIDlets

In [Chapter 1](#), we introduced you to MIDlets, applications that run on MIDP devices. MIDlets are written as one or more Java classes whose objects are compressed into a JAR file. Like Java applets, MIDP applications have an application life cycle while running on a mobile device. Specifically, a MIDlet can be in one of three states:

- Paused
- Active
- Destroyed

[Figure 3-1](#) shows the rules for transitioning between states.

Figure 3-1. MIDlet transition states



Here is a quick rundown of how MIDP applications change state: when a MIDlet is first started, it is placed in the paused state. After it's ready, the controlling software will then place the MIDlet in the active state. At this point, the MIDlet is running and the user can interact with it. The application can be placed back in the paused state by either the MIDP system or the program itself. In addition, the MIDP can be moved to the destroyed state from either the paused or the active state, again by either the MIDP system or the programmer. In the destroyed state, the MIDlet should release all of the resources it currently has back to the MIDP system.

We'll cover this in more detail in the following chapter, where we create and execute a MIDlet with multiple states. In the meantime, this quick introduction brings us to the point where we must first introduce some important concepts.

3.2.1 What Is a MIDlet Suite?

A *MIDlet suite* is simply two or more MIDlets that are packaged in a JAR file. MIDlets within the same suite can use the classes and resources contained in the JAR file, much like any standard Java application, the classes of which are loaded by the same class loader.

3.2.1.1 The JAR Manifest

The JAR file of a MIDlet suite often contains a manifest file with MIDlet attributes defined. These attributes describe the contents of the JAR file, which is in turn used by the application management software to identify and install the MIDlet suite. The attributes defined by the MIDP specification are listed in [Table 3-3](#).

Table 3-3. JAR manifest attributes		
Attribute name	Required	Description
MicroEdition-Configuration	Yes	The name and version of the J2ME configuration required. This uses the same format as the <code>MicroEdition.configuration</code> system property (for example, "CLDC-1.0").
MicroEdition-Profile	Yes	The name and version of the J2ME profile required. This uses the same format as the <code>microedition.profiles</code> system property (for example, "MIDP-1.0").
MIDlet- <i>n</i>	Yes	The name, icon, and class, separated by commas, of the <i>n</i> th MIDlet in the MIDlet suite.

MIDlet-Data-Size	No	The minimum number of bytes of persistent storage that the MIDlet requires. The default is zero.
MIDlet-Description	No	A description of the MIDlet suite.
MIDlet-Icon	No	The pathname of a PNG file within the JAR file to identify the MIDlet suite (not the individual MIDlets). It is used by the application management software to display an icon to identify the suite.
MIDlet-Info-URL	Yes	A pointer to a URL containing a detailed description of the MIDlet suite.
MIDlet-Name	Yes	The name of the MIDlet suite.
MIDlet-Vendor	Yes	The name of the organization (or vendor) providing the suite.
MIDlet-Version	Yes	The version number of the MIDlet suite presented in the format XX.YY.ZZ, where XX is the major, YY is the minor, and ZZ is the micro. If the micro is omitted, the default is zero. Therefore, the micro is optional. This information is used by the application management software for install and upgrade uses.

[Example 3-1](#) shows a sample manifest for a MIDlet suite (in this case, only two MIDlets) for a shopping MIDlet.

Example 3-1. A sample manifest

```

MIDlet-Name: ShopOnLine
MIDlet-Version: 1.0
MIDlet-Vendor: SELKOM
MIDlet-Description: a shopping MIDlet
MIDlet-Info-URL: http://www.selkom.com/shop
MIDlet-Data-Size: 500
MIDlet-1: BuyMIDlet, /icons/buy.png, com.selkom.BuyMIDlet
MIDlet-2: PayMIDlet, /icons/sell.png, com.selkom.SellMIDlet
MicroEdition-Profile: MIDP-1.0
MicroEdition-Configuration: CLDC-1.0

```

3.2.2 Java Application Descriptor (JAD)

Using a manifest to describe the MIDlets in the suite is a bit problematic. In [Chapter 1](#), we mentioned that before downloading a MIDlet or a MIDlet suite to a device, the Java Application Manager should check to make sure there is enough space for it. Using a manifest, however, means that the Java Application Manager should download the JAR file in order to read the manifest. Imagine downloading a MIDlet suite only to discover it cannot be installed on your device because it requires the next generation MIDP. To avoid these problems, the MIDP specification also defines the Java Application Descriptor (JAD).

A JAD file is a text file that is similar to a manifest. Unlike a manifest, however, it is not packaged in the JAR file. Similar to a manifest, it consists of a series of attributes used to describe a MIDlet suite. The possible attributes are shown in [Table 3-4](#).

Table 3-4. JAD attributes		
Attribute name	Required	Description
MIDlet-Name	Yes	The name of the MIDlet suite.
MIDlet-Version	Yes	The version number of the MIDlet suite. The format is XX.YY or XX.YY.ZZ, where XX is the major, YY is the minor, and ZZ is the

		micro that is optional. If the micro is omitted, the default is zero.
MIDlet-Vendor	Yes	The vendor of the MIDlet suite.
MIDlet-Jar-URL	Yes	The URL from which to download the MIDlet suite.
MIDlet-Jar-Size	Yes	The size of the MIDlet suite in bytes.
MIDlet-Description	No	A description of the MIDlet suite.
MIDlet-Icon	No	The pathname of a PNG file for the suite. The icon is used to identify the suite.
MIDlet-Info-URL	No	A URL that describes the MIDlet suite in greater detail.
MIDlet-Data-Size	No	The minimum number of bytes of persistent storage the MIDlet suite requires. If not specified, the default is zero.

As you can see from [Table 3-2](#) and [Table 3-3](#), there are some common attributes between the manifest and the JAD file. The mandatory attributes that must be duplicated in both the manifest and the JAD file are: MIDlet-Name, MIDlet-Version, and MIDlet-Vendor.

[Example 3-2](#) shows a JAD file for the same hypothetical MIDlet suite.

Example 3-2. A sample JAD file

```
MIDlet-Name: ShopOnLine
MIDlet-Version: 1.0
MIDlet-Vendor: SELKOM
MIDlet-Jar-URL:
http://www.selkom.com/shop/mid.jar
MIDlet-Jar-Size: 3544
MIDlet-Data-Size: 500
```

And that's the difference between a JAR manifest file and a JAD file in a MIDlet suite.

3.2.3 Programming Guidelines

Before we start programming with MIDlets, let's briefly discuss some guidelines that are useful when developing applications for mobile information devices such as cell phones and PDAs that you likely haven't considered before.

3.2.3.1 Performance

When programming for mobile devices with a small memory footprint, it is crucial to make your applications run faster. The less time your application takes to run, the happier your customers will be. For the J2SE programmer, here are some ways to help you achieve the best performance:

- Use local variables instead of fields. Accessing local variables is quicker than accessing class members.

- Minimize method calls. Remember that the Java virtual machine uses the stack to load and store a stack frame for every method it executes. For example, instead of doing something like:
- ```
for (int i=0; i<obj.getLength(); i++) {
```
- ```
    // do something with array elements
```
- ```
}
```

where the length of the array is evaluated every time through the loop, it is much more efficient to do this:

```
int len = obj.getLength();
for (int i=0; i<len; i++) {
 // do something with array elements
}
```

- Avoid string concatenation. This may cause a lot of object creation and subsequent garbage collection, and therefore decreases performance and increases the application's memory usage. It's often more efficient to use `StringBuffer` instead.
- Minimize object creation. Object creation leads to object destruction and reduces performance. Instead, design objects that can be recycled. Instead of creating return objects inside of methods, consider passing in a reference to the return object and modifying its values.
- Avoid synchronization. If an operation takes longer than a fraction of a second to run, consider placing it in a separate thread.

## Part II: Programming with the CLDC and the MIDP

Part II starts by elaborating on some of the concepts introduced in [Chapter 3](#). Later chapters show how to program with the CLDC/MIDP APIs, including GUI, event handling, networking, and databases. [Chapter 9](#) shows how to convert MIDlets into executable Palm applications for handheld devices running Palm OS v3.5 or higher.

[Chapter 4](#)

[Chapter 5](#)

[Chapter 6](#)

[Chapter 7](#)

[Chapter 8](#)

[Chapter 9](#)

## Chapter 4. Working with MIDlets