



C H A P T E R 4

A simple MIDP application

- 4.1 Questions about the MIDP development environment 56
- 4.2 Developing MIDP applications 56
- 4.3 Summary 68

This chapter introduces you to the entire process of creating a J2ME application using the Mobile Information Device Profile (MIDP). In order to illustrate this example a simple application will be used. In doing this, we introduce a number of J2ME terms and concepts, and provide a cursory introduction to the J2ME API. It's always a good idea to become familiar with some of the terminology and the paradigm of a new software environment before trying to tackle a big project. This will set the stage for upcoming chapters where each concept will be covered in more detail and we look at using J2ME to build our tutorial application. For now, the goal is to get an application up and running quickly and to introduce you to the MIDP development environment.

All of the examples are described using the Windows operating system. We do not address the particular syntax of other operating system commands, but the general concepts hold. If you are not running Windows, you will need to translate the commands appropriately.

4.1 *QUESTIONS ABOUT THE MIDP DEVELOPMENT ENVIRONMENT*

When starting out in any new application development environment, most people usually have a number of general questions about the environment and tools for doing the job. Let's see if we can head off a few of these before we get started.

4.1.1 *Can I do this without an actual device?*

Absolutely! Many emulators are freely available and allow you to run and test J2ME applications right on your desktop. We will discuss how to obtain and use each type of emulator when the time is right. But first we will concentrate on the code.

4.1.2 *What device do I start with?*

The Mobile Information Device Profile has been designed mainly with cellular phones and pagers in mind. However, MIDP can run on other types of devices, such as PDAs. Sun currently has an implementation of MIDP that runs on Palm OS devices. However, the current MIDP user interface capabilities are rather limiting on a PDA. For this example, a cellular phone will be chosen as the primary target device for the application. Since we are developing to the MIDP, rather than a specific device, the application will run on any MIDP-compliant device. So at this point, all we need to be concerned about is that the desired target devices support MIDP.

4.1.3 *Do I have to use the command line tools?*

No, there are a number of Integrated Development Environments (IDEs) available that take care of the dirty work for you. Sun's Wireless Toolkit is a good example. However, this chapter is intended to give you a detailed, behind-the-scenes example of what goes into creating a J2ME application. Therefore we will use the command line tools provided by Sun's reference implementations. We hope this will give you a better understanding of the technology.

4.1.4 *The example: what are we going to do?*

This chapter uses a variation of the ubiquitous Hello World application. The application is rather simple in functionality; it just displays a string of text to the screen. However, the intent of this chapter is to quickly cover the lifecycle of developing a complete application and deploying it to a device. More sophisticated applications will be built in later chapters.

4.2 *DEVELOPING MIDP APPLICATIONS*

As mentioned previously, this example will work for both a cellular phone and a pager. How does this dual functionality affect the way we write or build the application? As we will see, it does not affect how we create the application at all. The only difference comes at the end when we deploy the application and need to deal with the specific device itself.

Given the range of devices J2ME is designed to support, cellular phones and pagers rank at the low end, being two of the most limited devices in the J2ME spectrum. These limitations are especially noticeable in the areas of the user interface and available memory. Cellular phones, for example, typically have a one-handed keyboard. Entering letters becomes tedious quickly since the user is forced to cycle through three or more alphabetic characters represented on each key. Furthermore, cellular phones may have as little as 40 KB of memory available for your application once the virtual machine and runtime libraries are loaded.

As discussed in chapter 2, in order to deal with these limitations, both cellular phone and pager applications require a configuration and profile combination that addresses these limitations. This is where the Connected Limited Device Configuration (CLDC) and Mobile Information Device Profile (MIDP) come into the picture. The CLDC is designed for devices with limited characteristics. Since configurations address the horizontal needs of a wide variety of devices, an additional architectural piece is needed to support the more device-specific capabilities, most notably user interface and data storage. This is how profiles, or in this case MIDP, fits in.

Another piece that we will need is the virtual machine that supports the CLDC. This is the K virtual machine (KVM), which is also discussed in chapter 2. This is a specially designed reference implementation virtual machine that has a much smaller footprint than the standard Java virtual machine. Because of the small footprint, Java can run on memory-constrained devices such as a cellular phone.

4.2.1 Getting started

First we need to get our hands on the MIDP development environment. We will use Sun's reference implementation that is available in a single download from the following URL: <http://java.sun.com/products/midp>.

DISTRIBUTION NOTE As of this writing, the current publicly available version of the MIDP from Sun's web site listed above is version 1.0.3. However, depending on when you purchase this text and go to Sun's site, the version of MIDP may have changed. With the 1.0.3 release and using the default installation directories, MIDP installs in a directory called `midp1.0.3fcs`. This will obviously vary depending on your downloaded version. For this reason, we refer generically to the MIDP directory throughout this text as `midp-fcs`.

Download and unpack the distribution into the directory from which you want to work. Note that the distribution unpacks into a top-level directory named similar to `midp-fcs`. For convenience, set up the following system environment variables. These variables are used in this example for convenience and have no effect on the MIDP environment.

```
MIDP=\midp-fcs
MIDPClasses=\midp-fcs\classes
MIDPTools=\midp-fcs\bin
```

With the development environment in place, we are ready to begin developing our first J2ME application. Using MIDP, applications are created by extending the `javax.microedition.midlet.MIDlet` class. This class acts as the interface between the application management software on the device and MIDP applications. It is important to understand that each J2ME profile may define different starting points (classes and methods) for an application. For MIDP the starting point is a `MIDlet`.

4.2.2 What is a `MIDlet`?

A `MIDlet` is an abstract class that is subclassed to form the basis of the application. By subclassing the `MIDlet` class, we define an interface between our application and the application management software on the device. A `MIDlet` is the heart of a MIDP application and allows the device to start, pause and destroy the application.

The `MIDlet` class resides in the package `javax.microedition.midlet`. The code to declare a `MIDlet` looks something like this:

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

public class HiSmallWorld extends MIDlet {
}
```

For this example, we need to add a constructor that creates a `TextBox` (a GUI widget that allows us to display a message) and a member variable to hold the `TextBox` instance since we will need to reference it from a couple of places.

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

public class HiSmallWorld extends MIDlet {
    private TextBox textbox;
    public HiSmallWorld() {
        textbox = new TextBox("", "Hi Small World!", 20, 0);
    }
}
```

Since `MIDlet` is an abstract class, our `HiSmallWorld` class needs to implement a few methods before it will compile. There are three methods that require attention: `startApp()`, `pauseApp()` and `destroyApp(boolean unconditional)`.

When a device receives a message to start a `MIDlet`, the `MIDlet` is instantiated and the application management service on the device calls `startApp()`. At this point, our application takes over and does any initialization that may be required. In our example, we make the `textbox` the active element. Do not worry about the use of the `Display` class for now, as this will be covered in a subsequent chapter.

WARNING The `startApp()` method can be called a number of times during the lifecycle of a `MIDlet`. Therefore, it should not be used to perform initialization. For example, a `MIDlet` can be placed in a paused state as a result of a call to the `pauseApp()` method. In order to restart, and release it from the paused state, the `startApp()` method is invoked. If you have to do some initialization on the `MIDlet`, it needs to be carried out in conjunction with the constructor, not the `startApp()` method.

```
public void startApp() {
    Display.getDisplay(this).setCurrent(textbox);
}
```

The `pauseApp()` method is called by the device when the user, or the device, needs to suspend our application's activity to perform some other task. When the device invokes this method, our application is responsible for placing itself into a paused state.

Since we are only displaying a message to the screen, and there is nothing to do to pause the application, we will implement this as an empty method.

```
public void pauseApp() {
}
```

At the point, if the user chooses to close the application, or for some reason the system requests that the application be closed, the method `destroyApp(boolean unconditional)` is called. This method is invoked to allow our application to clean up any resources that it may be using, such as a network or database connection. This method takes a single, boolean parameter. This parameter indicates how much say our application has in being destroyed. If the parameter is true, our application will have no choice but to clean up its resources and prepare for being destroyed. If the parameter is false, the application can throw a `MIDletStateException` exception to prevent the `destroy` method from taking place and to continue running. Again, this exception can only be thrown if the parameter is false. Since there are no resources that need to be cleaned up in this application `destroyApp(boolean unconditional)` is also implemented as an empty method.

```
public void destroyApp(boolean unconditional) {
}
```

The full source code for our first J2ME application is shown in Listing 4.1.

Listing 4.1 HiSmallWorld.java

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

public class HiSmallWorld extends MIDlet {

    private TextBox textbox;
    public HiSmallWorld() {
        textbox = new TextBox("", "Hi Small World!", 20, 0);
    }

    public void startApp() {
        Display.getDisplay(this).setCurrent(textbox);
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
    }
}
```

This is all the code required to get our application up and running. The next step is to compile the application.

WIRELESS TOOLKIT Sun Microsystems provides an IDE for developing MIDP applications. Called the Wireless Toolkit, it is available from Sun's web site at: <http://java.sun.com/products/j2mewtoolkit/>. We do not use the toolkit throughout our examples and tutorial application for two reasons:

- 1 We want you to understand what is actually occurring behind the scenes when writing J2ME applications. The compiling, preverifying, jarring and deployment are important parts of the J2ME development process and should be understood.
- 2 IDEs change or may have bugs. You may switch development tools or you may find an IDE that has a problem or bug. An IDE can do part or most of the work for you when it comes to developing applications, but it is important to understand the work being accomplished by the IDE just in case the IDE has difficulties or you change IDEs.

In appendix D, we demonstrate the use of the Wireless Toolkit for the Hello World example. If you download the Wireless Toolkit from Sun, you should still be able to use the application code in the rest of this text. However, be aware that compiling, preverifying, jarring, and deploying of the applications will require different steps and use a different emulator executable.

4.2.3 Compiling the application

This is done using the standard `javac` compiler command. However, since we are compiling an application for the J2ME environment (rather than J2SE) the `-bootclasspath` option must be used. This option takes advantage of Java's cross-compilation capability. The cross-compilation feature is new in the Java 2 platform and allows the Java compiler to target the class files for an environment other than standard Java. Our target environment is J2ME and by using the `-bootclasspath` option we can instruct the compiler to use the J2ME libraries. Without this we could accidentally use classes or method signatures not supported by J2ME (such as `Double`) and as a result, these errors would not be caught until runtime.

Use the following command line to compile the application:

```
> javac -g:none -bootclasspath %MIDPClasses% HiSmallWorld.java
```

The `-g:none` option is used to prevent debug information from being included in the class files. This is an optional flag, but it helps reduce the size of the class files. The `%MIDPClasses%` variable is the environment variable we set up earlier. This variable points to the J2ME classes and it is passed as the `-bootclasspath` parameter.

4.2.4 Preverifying the application

For security reasons, the standard Java Runtime Environment verifies each class file before loading it into memory. This is done to ensure that the class file is valid and does not attempt to access memory outside of its boundaries or access disk. Since J2ME must cater to devices that are more limited than a desktop computer, some of the J2ME virtual machines handle class file verification somewhat differently than the standard Java VMs, namely, verification does not entirely take place on the device. Instead, as part of the deployment process, each class file must be preverified using a `preverify` utility provided in the J2ME development environment. This utility verifies each class file and modifies it to include special flags indicating their validity. At runtime, the J2ME virtual machine checks these flags. If the flags are present and indicate a valid class file, the VM assumes the class is OK to run. Without these flags the VM will throw an exception or abort the class loading process.

Preverification is performed using the `preverify.exe` utility found under the `bin` directory. Run the following command to preverify the application:

```
>%MIDPTools%\preverify -classpath %MIDPClasses%;. HiSmallWorld
```

It is important to note that this utility creates new class files. By default, this command places the output class files in a directory called `\output` off of the current directory. To change the output directory, use the `-d` option as with other Java utilities. The following version of the command places the class files in a directory named “preverified” nested below the current directory:

```
>%MIDPTools%\preverify -classpath %MIDPClasses%;.  
-d .\preverified HiSmallWorld
```

For each of these commands we specify a `classpath` of only the J2ME classes, and our own classes we have created, to ensure that the class files generated are suitable for the J2ME target environment.

If the `preverify` utility is having trouble loading your class file, which is reported by the message “Error loading class HiSmallWorld”, make sure `classpath` is set properly to find the file `HiSmallWorld.class` that was created by `javac`.

4.2.5 Running the application

With the classes compiled and preverified, our application is finally ready to run. This is where we need an emulator. If you downloaded the MIDP reference implementation, you already have an emulator and are ready to go. The emulator is an executable named `midp` and is located in the `midp-fcs\bin` directory. We will run our application by typing the following command:

```
>%MIDPTools%\midp -classpath %MIDPClasses%;.\output HiSmallWorld
```

This command runs the `midp` executable, passing the MIDP classes and our application’s classes on the `-classpath` parameter. Note that we must direct the `midp`

utility to look in the `.\output` directory (relative to the current directory) for the preverified version of our classes. If we had just specified the current directory (`.`), `midp` would find the original classes generated by `javac`. Since these classes do not contain the proper preverification flags, the J2ME runtime environment would not be able to load the classes and a runtime exception would abort the class loading process. If the application runs successfully, your emulator will look like figure 4.1.



Figure 4.1
The HiSmallWorld MIDlet written above is depicted here running in the MIDP emulator. While the MIDP specification dictates common Java functionality across the spectrum of devices, in this case cellular telephones, each device may have a slightly different display. Thus, emulators often provide various “skins” to test applications running in various displays.

After closing the emulator, the output from the console should look similar to the following text.

```
E:\work\HiWorld>\midp-fcs\bin\midp -classpath \midp-fcs\classes;.\output
HiSmallWorld
Execution completed successfully
8205 bytecodes executed
7 thread switches
204 classes loaded (149 bytes)
220 objects allocated (9572 bytes)
0 garbage collections
0 bytes collected
0 objects deferred in GC
0 (maximum) objects deferred at any one time
0 rescans of heap because of deferral overflow
0 pointer validations requiring heap scans
Current memory usage 9572 bytes
Heap size 300000 bytes
```

4.2.6 Troubleshooting

If there are problems running the application here are some debugging tips:

- Make sure the application compiled successfully when you ran `javac` and make sure the `preverify` utility ran successfully without errors.
- If an error such as “The name specified is not recognized as an internal or external command, operable program or batch file.” occurs, this means Windows was unable to find the `midp` executable. Adjust the command path to point to `midp.exe`.

- The most notorious runtime problem in the Java environment is getting the `classpath` set properly so that the correct versions of classes are loaded, and loaded in the proper order. The following two problems are related to `classpath`:
 - If an error such as “One or more MIDlet class(es) not found: null” was reported, the `midp` emulator was not able to find your classes. Make sure `classpath` is specified correctly and make certain your class files are where you think they are. Remember, the `classpath` must specify both the J2ME class libraries (`\midp-fcs\classes`) and your application’s classes.
 - If an error such as “ALERT: Error verifying class HiSmallWorld” was reported, the `midp` executable was unable to load the class. Most likely the emulator found the unverified version of `HiSmallWorld.class` instead of the preverified version, so make sure `classpath` includes the preverified version of the class. Be certain the unverified version is not included on `classpath` or its path is specified after the preverified path. Try deleting the unverified version of the class file to see if you get a different error or the correct, preverified version is found.

4.2.7 JARing MIDlets

The previous example shows the `midp` emulator directly accessing the class file. However, in most cases MIDP applications should be deployed as JAR files. This is done for several reasons. First of all, depending on the network protocol and the client-server software involved, JAR files can be more efficient when downloading multiple applications over protocols such as HTTP since the entire JAR is downloaded with a single connection (rather than a connection for each class file). Furthermore, MIDlets can be deployed as part of a MIDlet suite. The details of creating a MIDlet suite will be covered in a moment, but first we modify the example to use a JAR file for deployment.

Using the existing class files, we can run the following `jar` command to create a JAR file:

```
>jar cf hi.jar -C .\output HiSmallWorld.class
```

The “`cf`” parameters tell the `jar` utility to create a new JAR file named “`hi.jar`”. The `-C` option is used to change to a specified directory and include a specified file. In this case, the `-C` option is used to switch to the `\output` directory to pick up the `HiSmallWorld.class` file without having the `\output` directory appear in the JAR file as an attribute of the class. (Without using the `-C` option the runtime environment would think our MIDlet resided in a package named `output`.)

Now let’s run `midp` using our newly created JAR file. In order to do this, make a minor adjustment to the `classpath` setting to include the JAR file that now contains the class file.

```
>%MIDPTools%\midp -classpath %MIDPClasses%;.\hi.jar HiSmallWorld
```

This should not change the MIDlet. The only difference is that we are now running the application from a JAR file. If the emulator cannot find the class, then either the JAR file is not valid or there may be something wrong with the `classpath`.

4.2.8 Developing MIDlet suites

Multiple MIDlets can be grouped and deployed as a unit using a MIDlet suite. A MIDlet suite is composed of a JAR file containing all the MIDlets and supporting classes and an application descriptor file. The application descriptor file is a text file containing information about the MIDlet suite, such as the names of the MIDlets, the location of the JAR file, vendor information, etc. Application descriptor files have the extension “jad” and provide the device, and in some cases a server environment, with information about the MIDlet suite so it can be run over a network or installed physically on the device.

Deploying MIDlets as part of a suite has some advantages over deploying the MIDlets individually. The most significant advantage is that MIDlets in a suite can share resources such as data stored on the device. For example, within an MIDP implementation, records are stored in a device-dependent area that is not directly accessible by the Java APIs. This data storage area is controlled at the MIDlet level. Within a MIDlet suite however, all MIDlets can share record stores and create multiple, uniquely named, record stores. In addition to the ability to share resources, MIDlet suites are deployed using JAR files. As mentioned previously this can allow the client to be more efficient when downloading the application.

To better understand dealing with MIDlet suites, we are going to need more than one MIDlet. For simplicity, make a copy of `HiSmallWorld`, giving it the incredibly innovative name of `HiSmallWorld2` and change the output string to read “Hi Small World2”. Once this is done, compile and preverify the new `HiSmallWorld2` class.

```
>javac -g:none -bootclasspath %MIDPClasses% HiSmallWorld2.java
>%MIDPTools%\preverify -classpath %MIDPClasses%;. HiSmallWorld2
```

NOTE *Display limitations* It is worth pointing out that, on the MIDP cellular phone emulators, a 15-character `String` (give or take a few characters) is about the longest `String` that can be displayed without wrapping. Since the Connected Limited Device Configuration (which is the configuration for MIDP) addresses *limited* device implementations, care should be taken to understand the different limitations of the target devices for which you are writing applications. Different devices have different display limitations even though they all may support MIDP. Pagers, and other cellular phones, for example, may have a wider and narrower screen.

Now we are ready to create our MIDlet suite. There is no real significance to this suite in terms of functionality. The goal is to walk through how MIDlet suites are created.

The MIDlet suite descriptor file

The first step is to create a descriptor file for the MIDlet suite. A descriptor file is a text file with a `jad` extension. The attribute names are case-sensitive. A list of the attribute names and their purposes is provided in table 4.1. The Java Application Manager (JAM) on the device uses the descriptor to manage the application lifecycle. The JAM is responsible or participates in activities such as downloading, installing, inspecting, executing and uninstalling applications.

Table 4.1 The Java Application Descriptor is used by the JAM to manage a MIDlet suite's applications on the device. As this table shows, it contains a wealth of information about the suite.

Attribute Name	Description
MIDlet-Name	Name of the MIDlet suite.
MIDlet-Version	Version of the MIDlet suite. The format must follow the convention Major.Minor.Micro (X.X[.X]) where the micro version is optional (defaults to zero if omitted). Each version number is allowed two digits (0-99). If this tag is missing, the version is assumed to be 0.0.0. Any nonzero version is considered a newer version than 0.0.0.
MIDlet-Vendor	Vendor that supplies this MIDlet suite.
MIDlet-Description	Text description of the MIDlet suite. (Optional)
MIDlet-Info-URL	Location where more information can be found about the suite. (Optional)
MIDlet-Jar-Size	Size of the JAR file specified by this descriptor.
MIDlet-Jar-URL	The URL indicating from where the JAR can be loaded.
MIDlet-Data-Size	The minimum number of bytes of persistent data required by the MIDlet suite. The default is zero. (Optional)
MIDlet-Icon	The name of a portable network graphic file (PNG) within the JAR file representing the MIDlet suite. (Optional)
Micro Edition-Profile	Profiles used by the application.
Micro Edition-Configuration	Configuration used by the application.
MIDlet-1	The first MIDlet in the list of available MIDlets (if this is a MIDlet suite). For each MIDlet specified, the following syntax is observed: <i>Description, icon name, MIDlet class name.</i>
MIDlet-n	The description appears in the menu when the list of MIDlets is displayed. Nth MIDlet in the suite

For our example we define a JAD (Java Application Descriptor) file with the following properties. We do not specify an icon for any of our MIDlets at this point. Create this file in the current directory. If you have been following the examples, this is the same directory where the Java source files you are working with are located.

```
MIDlet-Name: SmallWorldsuite
MIDlet-Version: 1.0.0
MIDlet-Vendor: Catapult Technologies, Inc.
MIDlet-Description: Sample suite of Small World MIDlets
MIDlet-Info-URL: http://www.ctimn.com/
MIDlet-Jar-URL: http://localhost/hi.jar
MIDlet-Jar-Size: 3000
MicroEdition-Profile: MIDP-1.0
```

```
MicroEdition-Configuration: CLDC-1.0
MIDlet-1: Hello1, , HiSmallWorld
MIDlet-2: Hello2, , HiSmallWorld2
```

JARing the MIDlet suite

The JAR file for a MIDlet suite must contain a manifest. A manifest provides the runtime environment information about how the JAR file is configured, any security information and what the JAR contains. The J2ME runtime environment compares the manifest to the application descriptor as a precaution before loading a MIDlet suite.

The values of `MIDlet-Name`, `MIDlet-Version` and `MIDlet-Vendor` must be the same in both the manifest and the descriptor file. If these values do not match, the MIDlet suite is considered invalid. Developers may define descriptor attributes not beginning with `MIDlet-` to provide property information to the application.

To create a manifest, simply provide the JAD file as input to the `jar` command. Modify the `jar` command used previously to create the JAR for our MIDlet suite.

```
>jar -cfm hi.jar HiMIDletsuite.jad -C ./output HiSmallWorld.class -C
./output HiSmallWorld2.class
```

The `jar` command now contains an “m” option instructing the JAR utility to create a manifest using `HiMIDletsuite.jad`. Note that the `-C` option must be repeated for each class specified in the `jar` operation. If a wildcard is used (e.g., `*.class`) the `-C` option is applied only to the first class file and is ignored for the remaining class files. In this scenario, unverified classes can accidentally be added to the JAR file, causing problems at runtime.

Now we are ready to run our MIDlet suite. Use the `-descriptor` option with the `midp` emulator to run the suite directly.

```
%MIDP%\bin\midp -classpath %MIDPClasses%;.\hi.jar -descriptor HiMIDletsuite.jad
```

The first screen that appears is a list of our MIDlets that make up the MIDlet suite. This list is composed of the MIDlet description specified for each MIDlet in the JAD file. At this point, we do not have an Exit button defined that allows the user to exit the application gracefully. This requires a user interface component and the use of event handling that is beyond the scope of this exercise. Both user interface components and event handling are covered in the next chapter. For now, we have to live with running one MIDlet at a time and exiting the emulator. Figure 4.2 shows the Hello2 MIDlet running.

So now we have managed to build and run MIDlets and MIDlet suites. How are MIDlets intended to be used in the real world? So far we have been running MIDlets directly from the computer on which we develop them, using the `midp` emulator. However, running MIDlets on an actual device is slightly different. First of all, the application must somehow get onto the device. There are basically two ways that MIDlets can find their way onto a device. They can be installed physically to the device or they can be temporarily loaded into memory over a network connection.

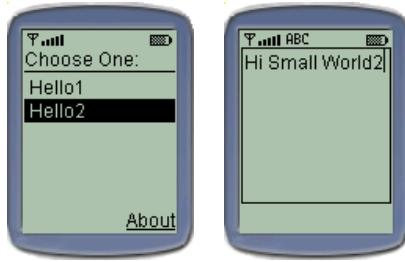


Figure 4.2 HiSmallWorld2 joins HiSmallWorld as part of the HiMIDletsuite running in the MIDP emulator. As the picture on the left shows, when a MIDlet suite is deployed to a device, the device knows to provide an application kick-off screen that allows the user to select MIDlets for execution.

Fortunately, the `midp` emulator supports the ability to run MIDlets in both of these ways in addition to running them directly, as we have done so far. To begin, we discuss simulating MIDlet deployment using the `midp` emulator. This allows us to explore these deployment techniques and get our environment set up correctly. Once these concepts are familiar to us, we will deal with the actual devices.

Accessing MIDlets over the Internet is a very likely scenario so we will begin by accessing our MIDlet suite using a Web server. In this scenario the application is dynamically downloaded to the device each time we run the emulator.

4.2.9 Running MIDlet suites from a web server

In order to access a MIDlet using a Web server, you need a Web server that the `midp` emulator can access. This example uses the Apache web server, which is available at the following URL: <http://httpd.apache.org>

Once the Web server is installed, the MIME type configuration needs to be modified to handle the `jad` extension. MIME stands for Multipurpose Internet Mail Extension and allows the Web server to know what types of content the client supports.

For Apache, adding the following line to the `mime.types` file specifies the JAD MIME type.

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

Deploying a MIDlet suite to a web environment is simply a matter of placing the JAR and JAD files in an area visible to the Web server. For Apache, this is the `htdocs` directory. Copy the files `hi.jar` and `HiMIDletSuite.jad` into this directory and start the web server. Make sure the Web server starts without errors. Then invoke `midp.exe` using the `-transient` option.

```
>%MIDP%\bin\midp -transient http://localhost/HiMIDletSuite.jad
```

There should not be any differences in the application itself. The only difference is that we are now accessing the application over `http`.

4.2.10 Installing MIDlet suites locally

The `midp` emulator supports the ability to emulate installing a MIDlet suite from a location, either a file or URL, so we can run it locally on the “device.” The following command simulates installing a MIDlet suite locally on a device via a Web server.

This command assumes the Web server is up and running and the application has been deployed to an area visible to the Web server. (See the previous example to understand how to set this up.)

```
>%MIDP%\bin\midp -install http://localhost/HiMIDletSuite.jad
```

Before we run the installed suite, let us make sure our application is installed. This can be done using the `-list` option:

```
>%MIDP%\bin\midp -list
```

The output should be something like the following:

```
E:\_book\work\HiWorld> \_book\midp-fcs\bin\midp -list
JamMode = LIST
  SmallWorldSuite
    Hello1
    Hello2
```

Once a MIDlet suite is installed, the Web server is no longer necessary. The application can run as if physically installed on the device using the `-run` option:

```
>%MIDP%\bin\midp -run SmallWorldSuite
```

Note that the `-run` option requires the name of the suite specified in the JAD file, not the name of the JAD file.

To remove an installed MIDlet suite use the `-remove` option followed by the name of the suite to remove:

```
>%MIDP%\bin\midp -remove SmallWorldSuite
```

To obtain profile and configuration information for an installed suite, use the `-version` option followed by the name of the suite:

```
>%MIDP%\bin\midp -version SmallWorldSuite
```

4.3 SUMMARY

In this chapter, we have looked at setting up a J2ME development environment, specifically a MIDP environment. We also examined a little of the CLDC and MIDP API while developing the simplest of applications. With the development environment in place and a fundamental understanding of how J2ME applications are built and deployed using MIDP, you are ready to get into some of the more powerful capabilities of this J2ME environment.